FUNDAMENTOS DO PENSAMENTO ALGORÍTMICO
NATANAEL ANTONIOLI

1 SUMÁRIO

2 PREFÁCIO

Bem-vindo! Antes de seguirmos no curso, considero prudente me apresentar como autor. Chamo-me Natanael Monteiro Pastore Antonioli, sou estudante de Engenharia da Computação pela UFSCar e dono do canal no YouTube Fábrica de Noobs, no qual procuro, entre outros assuntos, apresentar cursos técnicos e teóricos de Computação.

Nesta apostila, intitulada Fundamentos do Pensamento Algorítmico, procuro apresentar ao leitor as bases da programação estruturada, fundamental para qualquer ingressante na Computação. No futuro, pretendo transformar essa apostila em um curso em vídeo, que será postado em meu canal. Então, não deixe de acessá-lo e verificar se tal produção já começou.

O pensamento algorítmico é, sem dúvidas, a base de conhecimento fundamental para qualquer estudante que deseja se iniciar no mundo da programação de computadores e da informática, uma vez que é responsável pela capacidade do programador de representar a solução de um problema na forma de uma sequência de instruções claras e objetivas que poderão ser posteriormente reproduzidas pela máquina.

Dessa forma, o objetivo dessa apostila não é ensinar a programação em alguma linguagem específica – isso será feito posteriormente – mas sim introduzir no estudante toda a base de conhecimento necessária para, no momento de programar, ser capaz de dominar todas as estruturas existentes e saber aplica-las com precisão.

Entretanto, do ponto de vista pedagógico, não considero adequado que o leitor “devore” todo o conteúdo apresentado nessa apostila sem, no decorrer deste processo, aplica-lo em alguma linguagem de programação de sua preferência, uma vez que grande parte dos conceitos aqui apresentados nos capítulos finais são utilizados apenas em estruturas mais avançadas da programação formal e, por essa razão, dificilmente serão bem aproveitados de primeira.

Como esse material é voltado para auxiliar estudantes ingressantes em cursos de Computação, alguns formalismos um tanto desnecessários para entusiastas serão abordados. Todavia, essa apostila pode ser aproveitada por qualquer tipo de público e, para tanto, procurarei esclarecer quais conceitos são fundamentais e quais são dispensáveis.

Em cada capítulo, procurei apresentar um problema prático cuja solução envolve dominar o conceito abordado no capítulo, o qual será discutido e, ao final,

será utilizado na construção de uma solução algorítmica para o problema de abertura.

Eventuais dúvidas podem ser publicadas em nosso canal no Reddit, em https://www.reddit.com/r/fabricadenoobs. Lá, selecione a *flair* para Lógica de Programação e compartilhe sua dúvida para discussão.

Bons estudos!

3 ALGORITMOS: O QUE SÃO, PARA QUE SERVEM

3.1 ALGORITMO NO DIA-A-DIA

Mesmo que você, em toda a sua existência, nunca tenha ouvido a palavra "algoritmo", certamente já criou e utilizou um, mesmo sem conhecer. Por exemplo, suponha que você está dirigindo em um bairro desconhecido e peça ajuda a alguém para chegar em determinado lugar.

Essa pessoa, provavelmente, irá lhe fornecer uma sequência de passos, do tipo "siga em frente por três ruas, depois vire à direita, siga por mais quatro ruas e vire à esquerda". Tal sequência de passos é, justamente, um algoritmo, uma vez que lhe fornece as instruções necessárias para resolver um problema específico.

Tal sequência de passos pode ainda ficar mais complexa: talvez, em determinado ponto, você tenha que verificar se uma rua está ou não liberada para acesso e, caso estiver, seguir por ela, de forma a ser necessário analisar situações do ambiente variáveis para tomar decisão – em suma, realizar um raciocínio.

Aqui, já podemos extrair parte da definição de algoritmo: trata-se de um conjunto de regras, instruções e raciocínios que permitem chegar a um objetivo.

Além disso, essas instruções são suficientes para que você seja capaz de entendê-las e executá-las. Entretanto, se esse mesmo algoritmo fosse entregue a uma criança, ele não seria tão facilmente interpretado e talvez nem seria executado.

Agora, imagine uma sequência de instruções presentes em um artigo científico a respeito de como isolar uma cultura de bactérias. Evidentemente, seriam empregados diversos termos técnicos desconhecidos pelo público leigo, o que não seria um problema, uma vez que os leitores de tal artigo, provavelmente, já conhecem tais termos.

Ou seja, temos outra característica importante de um algoritmo: suas instruções de adequam à pessoa ou público responsável por executá-lo.

Por fim, voltemos ao exemplo que iniciou esta sessão. Dificilmente tais instruções viriam com detalhes do tipo "se o semáforo estiver vermelho, aguarde". Isso acontece por que se espera que o leitor do algoritmo saiba como um semáforo funciona e como reagir diante dele, sendo desnecessário apresentar tais instruções. Entretanto, um pai, ao entregar o carro pela primeira vez ao filho, provavelmente iria incluir tais instruções.

Apesar deste exemplo se parecer bastante com a característica citada anteriormente, há algo a mais para se destacar: o nível de detalhamento do algoritmo também varia conforme a pessoa ou público responsável por executá-lo.

Além disso, caso a pessoa que lhe apresentasse termos dúbios, tais como “vire para cá”, você certamente iria desejar uma instrução mais clara. Isso acontece por que um algoritmo deve possuir passos claros o suficiente para não produzirem dúvida.

Por fim, é esperando que tal sequência de instruções termine – afinal, ninguém deseja ficar eternamente repetindo a mesma sequência de passos em círculos. Daí, tiramos a característica final: é necessário que o algoritmo seja finito.

Aqui, já podemos apresentar uma definição formal de algoritmo que engloba todos os critérios elencados anteriormente:

“Um algoritmo é um conjunto finito de instruções precisas que, se executadas, permitem a manipulação de um conjunto finito de dados de entrada para produzir um conjunto finito de dados de saída, dentro de um tempo finito.”

Na vida cotidiana, podemos encontrar exemplos de algoritmos em manuais de instruções, receitas culinárias, aparelhos de GPS, entre outros.

3.2 ALGORITMO NA COMPUTAÇÃO

Apesar dos exemplos que utilizamos no capítulo anterior, há uma diferença entre os algoritmos da vida cotidiana e os algoritmos que encontramos na computação, principalmente no que tange à forma como os procedimentos são declarados.

Para tanto, suponha um algoritmo para multiplicar dois números, revelar o resultado da operação e dizer se ele é negativo ou positivo. Até agora, temos algoritmos como meras sequências de instruções e, portanto, uma opção válida para construir tal algoritmo é a seguinte:

```
Obter os dois valores.
Multiplicar os dois valores
Se for positivo, informar que é positivo e se for negativo, informar que é negativo.
Informar o resultado
```

De fato, tal sequência de instruções atende todos os requisitos que estudamos, e alguém com capacidade mental razoável dificilmente teria problemas em entende-la.

Entretanto, suponha agora que você precise solucionar um problema mais complexo, tal como “receber um valor determinado pelo usuário de números inteiros, imprimi-los em ordem crescente e calcular a média de tais valores” ou, ainda, que precise imprimir os primeiros n dígitos da Sequência de Fibonacci.

Definitivamente, um algoritmo composto de centenas de passos, envolvendo instruções do tipo “volte para o passo tal até tal coisa acontecer”, “some tal número com tal número, e depois volte para tal passo” seria de difícil compreensão.

Imagine, agora, que outra pessoa escrevesse um algoritmo nesse estilo e coubesse a você implementá-lo em alguma linguagem de programação, ou verificar se ele atende ao problema proposto. Definitivamente, é algo enlouquecedor.

Por essa razão, na Computação, o conceito de algoritmo engloba algumas características adicionais. São eles:

1. Certo formalismo, normalmente convencionado previamente.
2. Certo padrão visual, voltado para facilitar a compreensão.
3. Um conjunto de estruturas lógicas que possuem equivalentes em linguagens convencionais, utilizadas para a criação dos procedimentos algorítmicos.

Se você está aprendendo programação de forma autônoma, isto é, sem compromissos com algum instrutor – e, principalmente, sem provas – pode ser dar ao luxo de dispensar (até certo ponto, claro) os itens 1 e 2 (principalmente o item 1), atentando-se ao item 3, que será o mais importante para a prática de programação.

Entretanto, se você está se iniciando em algum curso na área da Computação, provavelmente precisará se atender aos três itens e, por mais decepcionante que isso pareça, ao item 1.

Porém, tais itens estão tão interligados que dificilmente poderiam ser abordados de forma separada. Por essa razão, irei guiar as explicações com base nos tópicos presentes no item 3, mas abordando os outros dois aspectos quando se fizer necessário.

3.3 CONCEITOS PRELIMINARES

Nosso foco é a construção de algoritmos que, posteriormente, norteiem programas para serem executados em computadores. Por essa razão, é fundamental que nossos algoritmos criados tenham algumas características comuns aos programas de computador.

Dessa forma, iremos idealizar um "modelo de computador", que será um computador fictício responsável por "executar" nossos algoritmos. Na prática, tal modelo é apenas uma versão muito simplificada de um computador real, de forma que possamos simular algumas características deste.

O modelo utilizado no decorrer deste curso é o seguinte:
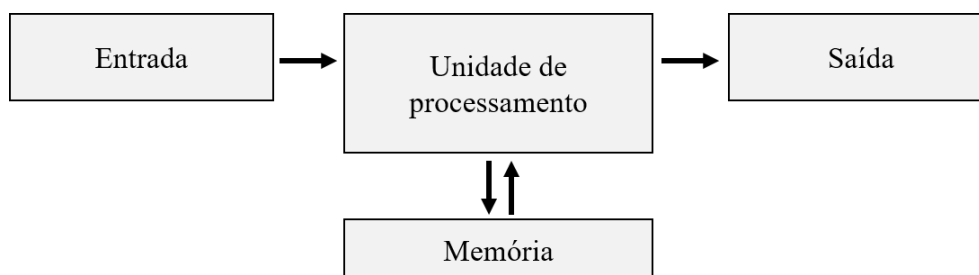
Figura 1: Modelo de computador utilizado neste curso.

Em suma, ele permite que as seguintes operações sejam realizadas:

1. Recebimento de valores de entrada, provenientes do usuário.
2. Processamento desses valores e execução do algoritmo.
3. Armazenamento de dados.
4. Retorno de valores de saída, exibidos para o usuário.

Você provavelmente já foi capaz de, intuitivamente, relacionar essas 4 operações com cada um dos componentes do nosso modelo de computador.

Mas, caso contrário, as caixas de entrada e saída servem para receber e exibir valores ao usuário, equivalentes a um teclado/mouse e um monitor, respectivamente. A unidade de processamento, evidentemente, corresponderia ao processador.

Entretanto, o item que mais merece atenção é o último, a memória, cuja função é armazenar dados. Em um computador, essa tarefa é realizada por diversos componentes, que podem ser divididos em memória volátil e memória não volátil.

A principal característica da memória não volátil é seu armazenamento a longo prazo: dados gravados aqui não são apagados quando a energia é cortada, por exemplo. Entre os componentes da memória não volátil, podemos citar os discos rígidos.

Já na memória volátil os dados armazenados são de curto prazo: eles não ficam armazenados permanentemente, aguardando serem acessados. É o exemplo da memória RAM, que recebe informações para serem usadas naquele momento e, após isso, descartadas.

A maioria dos algoritmos que criarmos terá como memória a memória volátil, utilizada para armazenamento das variáveis.

3.4 VARIÁVEIS

Mas, o que são variáveis?

Suponha que, em um algoritmo, seja necessário receber dois valores e multiplica-los entre si, tal como no primeiro exemplo desta apostila. Além disso, é necessário elevar o resultado dessa multiplicação ao primeiro dos valores e, depois, tirar a raiz quadrada.

Evidentemente, chamar o primeiro e o segundo valor de valor_1 e valor_2, respectivamente, seria de grande ajuda para facilitar a compreensão do algoritmo. Talvez, até pudéssemos chamar o resultado da multiplicação de multi, e o resultado da exponenciação de exp.

Nesse exemplo, valor_1, valor_2, multi e exp foram nomes atribuídos a uma “estrutura” capaz de armazenar valores.

É justamente a essa “estrutura” que damos o nome de variável. Vamos para a definição formal:

“Na programação, uma variável é um objeto (uma posição, frequentemente localizada na memória) capaz de reter e representar um valor ou expressão. ”

Outra forma mais intuitiva e didática de representar variáveis, muito utilizada em cursos de Computação, é defini-las como “caixas” que recebem valores provenientes do usuário ou da operação entre outras variáveis.
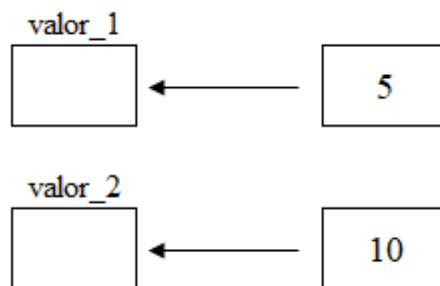
Figura 2: representação gráfica de duas variáveis

3.5 TIPOS DE DADOS

Já vimos que a melhor forma de lidar com dados é através de variáveis. Sendo assim, irei contextualizar uma situação que demonstra a necessidade da definição formal de alguns tipos de dados que as variáveis podem receber.

Primeiro, pense na sua idade e, em seguida, em há quantos anos você teve seu primeiro contato com um computador. Agora, subtraia esses dois valores. O resultado foi outro número, certo?

Agora, pense na sua idade e, em seguida, no seu nome. Agora, subtraia esses dois valores. Se o universo não explodiu, você provavelmente não foi capaz de realizar essa operação. Por que isso acontece?

Ora, a resposta é evidente. Seu nome é uma palavra e a sua idade é um número e, claro, não podemos somar números com palavras. Mas como explicar isso a um computador?

Para tanto, formalizamos que a variável responsável por receber a sua idade e a variável responsável por receber seu nome são de tipos diferentes e, por essa razão, não podem ser associadas.

Na construção de algoritmos, convencionamos cinco tipos de variáveis que atendem a – quase – todos os tipos de problemas. Nas próximas linhas, abordaremos esses tipos.

A primeira delas é a variável do tipo **inteiro**. Como o próprio nome indica, uma variável do tipo inteiro pode receber números inteiros e permite a realização das operações no campo dos inteiros.

Além disso, temos variáveis **reais** que, como o nome também indica, recebem números reais e permitem a realização de operações no campo dos reais.

Para o armazenamento e manipulação de valores textuais, como nomes, códigos alfanuméricos e textos com e sem pontuação, existe o tipo denominado **literal**. Dessa forma, uma variável literal sempre corresponde a uma cadeia de caracteres.

Outro tipo de dados bastante utilizado é o tipo **lógico**, que recebe apenas dois valores: verdadeiro ou falso, muitas vezes identificados como 1 e 0, respectivamente. Esse tipo de dado é bastante utilizado em condições, que serão estudadas posteriormente.

Por fim, existe o **ponteiro**, que, por envolver conceitos mais complexos, será abordado apenas em capítulos seguintes

3.6 REFINAMENTOS SUCESSIVOS

Quando nos deparamos com um problema muito extenso, é comum que não saibamos exatamente como começar a solução algorítmica para resolvê-lo. Nesse caso, muitos programadores recorrem à técnica de refinamentos sucessivos, isto é, o processo de escrever a mesma sequência de passos diversas vezes, mas cada vez mais detalhada.

Abaixo, há o mesmo algoritmo, escrito de duas distintas, através da técnica de refinamentos sucessivos. Todas as soluções se aplicam ao mesmo problema: verificar se uma equação de segundo grau possui raízes reais, dado os coeficientes e o termo independente.

O último código está escrito na linguagem algorítmica formal, que abordaremos no próximo capítulo. Aproveite para se familiarizar com ela.

```
Ler os valores dos coeficientes e do termo independente

Calcular o discriminante

Com base no sinal do discriminante, informa quantas raízes existem.
```

```
Ler a, b, c  sendo a e b os coeficientes e c o termo independente

determinante = b² - 4 * a * c

Se determinante ≥ 0:

        Dizer que tem raízes reais

Se determinante < 0:

        Dizer que não tem raízes reais
```

```
Algorithim Existência de raízes
Input: coeficientes e termo independente de uma equação de segundo grau
Output: se a equação possui ou não raízes reais
► Declaração das variáveis utilizadas
        declare:
                a,b,c: integer
                discriminante: real
► Leitura dos dados de entrada
        read a, b, c
► Cálculo do discriminante
        discriminante ← b² - 4 × a × c
► Escrita dos dados de saída
        if discriminante > 0 then
                write "A equação de segundo grau possui raízes"
        else
                write "A equação de segundo grau não possui raízes"
        end if
end Algorithim
```

4 SINTAXES FORMAIS EM ALGORITMO

Finalmente, é chegado o momento de compreendermos como nossos algoritmos serão criados.

4.1 CABEÇALHO

Conforme estudado nos capítulos anteriores, vimos que algoritmos tem um objetivo (informar um valor, ou imprimir determinadas informações, por exemplo), valores de entrada (os quais serão inseridos pelo usuário – e podem muito bem ser nulos) e valores de saída (os quais serão exibidos para o usuário).

Esses três tópicos são os primeiros que devem ser declarados ao se criar um algoritmo.

Para especificar o objetivo do algoritmo, utilizamos o termo **Algorithim:** seguido de um breve título a respeito do objetivo do algoritmo. Detalhe: não é necessário especificar, nessa parte, detalhes como input e output, uma vez que eles serão prontamente especificados nos próximos tópicos. Abaixo, há alguns bons exemplos.

Algorithim: Fatorial de um número

Algorithim: Sequência de Fibonacci

Algorithim: Lista de candidatos

Em seguida, precisamos especificar quais valores são inseridos pelo usuário e quais serão retornados pelo algoritmo. Para tanto, usamos os termos **Input:** e **Output:** seguidos da descrição a respeito dos valores de entrada e saída, respectivamente. Observe, abaixo, alguns exemplos:

Algorithim: Fatorial de um número

Input: número inteiro

Output: fatorial desse número

Algorithim: Lista de candidatos

Input: nome e nota de *n* candidatos

Output: quais candidatos foram aprovados

Algorithim: Sequência de Fibonacci

Input: não há

Output: primeiros 15 números da sequência de Fibonacci

4.2 RODAPÉ

Ao final do programa, basta encerrarmos o algoritmo com **end Algorithim**.

4.3 COMENTÁRIOS

Às vezes, no decorrer do algoritmo, surge algum procedimento que relativamente complexo, que necessitaria de alguma explicação adicional a respeito de como determinado processo funciona.

É justamente nessas ocasiões que utilizamos o indicador ►, o qual sempre é seguido de alguma explicação a respeito de um procedimento próximo a ele. Outra utilidade para comentários é separar partes do código, a fim de mantê-lo mais organizado.

Observe, abaixo, alguns exemplos:

```
► Escrita dos dados de saída
    if discriminante > 0 then
        write "A equação de segundo grau possui raízes"
    else
```

```
► Contagem até 10
    for i ← 0 to 10 step 1 do
        a ← a + 1 ► Adiciona 1 ao valor de a
        write a
    end for
```

5 OPERADORES DE VARIÁVEIS

5.1 ATRIBUIÇÃO

Já estudamos, anteriormente, qual a função de variáveis: variáveis armazenam valores, que podem ser de cinco tipos e serem oriundos do usuário ou de processos do próprio algoritmo.

Também sabemos que variáveis possuem nome, e atribuir nomes para variáveis é fundamental em qualquer algoritmo. Mais do que isso, é importante que o nome da variável esteja relacionado com o seu conteúdo.

Entretanto, antes de operar as variáveis, é necessário declará-las: isto é, informar ao programa quais são seus tipos. Para tanto, utilizamos a estrutura **declare:**, seguida, nas próximas linhas, do nome da variável, dois pontos e seu tipo, que possui as seguintes formas: **integer** para inteiro, **float** para real, **string** para texto e **boolean** para lógico.

Quando temos várias variáveis de um mesmo tipo, também podemos separá-las por vírgula, caso desejado. Recomendo utilizar essa notação apenas caso essas variáveis estejam relacionadas.

Observe, abaixo, alguns exemplos:

```
declare:
	a,b,c, discriminante: integer
```

```
declare:
	a: integer
	b: real
```

```
declare:
	nome: string
	nota: real
	aprovado: boolean
	total_alunos: integer
	i: integer ► contador
```

É apenas depois de declaradas que as variáveis podem ser operadas. Dentre as operações, a mais elementar é, provavelmente a atribuição de um valor, que serve para todos os tipos de variáveis.

Tal atribuição é feita utilizando o operador ←, mantendo a variável em questão à esquerda do operador e o valor a ser recebido à direita dele. Variáveis podem receber valores prontos, valores vazios, operações de outras variáveis e inputs do usuário (abordado na próxima seção).

Observe, abaixo, alguns exemplos, nos quais as variáveis já foram anteriormente declaradas:

```
a ← -2

b ← 5.3
```

```
nome ← "Fábrica de Noobs"

milhão_inscritos ← 1
```

Nesse processo, algumas regras devem ser seguidas. São elas:

- Variáveis do tipo *string* recebem apenas valores entre aspas.
- Variáveis do tipo *boolean* recebem apenas 1 e 0.

Além disso, existem casos em que variáveis, implicitamente, devem começar com o valor zero, como por exemplo uma variável que receberá uma soma de valores a partir do 0. Nesses casos, atribuímos valor vazio à variável com o símbolo ∅. Observe:

```
soma_valores ←∅
```

5.2 ENTRADA

Na maioria dos programas, é comum que variáveis recebem valores inseridos pelo usuário. Logo, é necessário que seja possível indicar tal operação em um algoritmo.

Em pseudocódigo, realizamos essa operação por meio do comando read, seguido da variável em questão ou de um conjunto de variáveis separadas por vírgula. Não é necessário inserir nenhum tipo de mensagem solicitando ao usuário a inserção da variável, uma vez que isso fica implícito no algoritmo. Por exemplo:

```
read a
```

```
read nota_1, nota_2, nota_3
```

5.3 SAÍDA

Em alguns momentos, também é necessário informar algo para o usuário. Normalmente, essa situação pode ser de dois tipos.

Para inserirmos uma mensagem simples, sem nenhuma variável, basta utilizarmos o comando write, seguindo da mensagem, entre aspas. Por exemplo:

```
write “Morte ao Miojoo”
```

Já para inserirmos uma mensagem contendo uma variável, temos duas opções. A primeira delas resume-se em um write seguido da variável em questão, ou de um conjunto de variáveis, separadas por vírgula. Costumamos utilizar essa sintaxe quando o objetivo da mensagem é apenas informar as variáveis, sendo possível, portanto, a omissão de demais palavras presentes nela. Por exemplo:

```
write media, frequência
```

A outra possibilidade envolve um write seguido da mensagem entre aspas, sendo interrompida por variáveis quando necessário. Utilize essa sintaxe apenas quando a mensagem contiver texto que não é implícito pela variável. Observe abaixo:

- Aqui, a parte “e, portanto, ele está aprovado” não está indicada explicitamente e, portanto, deve ser indicada.

```
write “a média do aluno é”, media, “com frequência de”, frequência, “e, portando, ele está aprovado”
```

- Aqui, o texto não é necessário, uma vez que não contém nenhuma informação adicional.

```
write “a média do aluno é”, media, “com frequência de”, frequência
```

- Portanto, deve-se usar a sintaxe:

```
write media, frequencia
```

5.4 OPERADORES MATEMÁTICOS E DE TEXTO

Como mencionamos anteriormente, podemos também realizar operações entre variáveis, que se dividem entre as operações aritméticas, literais, relacionais e lógicas.

Dentre as operações aritméticas, temos as operações usais, normalmente convencionadas como:

Função	Recebe	Retorna	Descrição
a+b	Real	Real	Adição
a-b	Real	Real	Subtração
a/b	Real	Real	Divisão
a×b	Real	Real	Multiplicação
a ^ b	Real	Real	Potenciação
a++	Real	Real	Incremento
a--	Real	Real	Decremento

Tabela 1: operadores aritméticos comuns em algoritmos.

E também as funções, que normalmente recebem apenas um valor. Abaixo, estão algumas das principais funções.

Função	Recebe	Retorna	Descrição
sen (r), cos (r), tan(r), asen(r), acos(r), atan(r)	Real	Real	Funções trigonométricas usuais
ln (r), log (r)	Real	Real	Funções logarítmicas usuais
trunca(r)	Real	Inteiro	Retorna a parte inteira de um número real
arred(r)	Real	Inteiro	Retorna o inteiro mais próximo do real passado
abs(r)	Real	Real	Módulo (valor absoluto) de um número real
resto(r,i)	Real/inteiro	Inteiro	Retorna o resto da divisão entre dois números
quoc(r,i)	Real/inteiro	Inteiro	Retorna o quociente da divisão entre dois números
raiz(r)	Real/inteiro	Real	Raiz quadrada de um número

Tabela 2: funções matemáticas comuns em algoritmos.

Quando estamos lidando com strings, podemos também aplicar algumas funções. São elas:

Função	Recebe	Retorna	Descrição
s + r	String	String	Concatena duas strings.
comprLiteral(s)	String	Inteiro	Comprimento de uma string.
valLiteral(s)	String	Real	Retorna o valor numérico de uma

	cadeia de caracteres.

Tabela 3: funções de texto comuns em algoritmos.

E o que fazer quando você precisa utilizar uma função que não está listada acima? Como algoritmo não é compilado por nenhum programa de computador, temos duas possibilidades.

A primeira delas envolve criar uma função e escrever todas as instruções que ela deve executar, conforme as instruções que você encontrará daqui alguns capítulos.

A segunda envolve criar uma função, mas não necessariamente especificar todas as instruções, apenas inserindo um comentário a respeito do papel de tal função. Entretanto, você só deve utilizar esse recurso quando a função em questão foge do escopo do algoritmo, ou quando se trata de uma função comumente encontrada em pacotes auxiliares das linguagens de programação.

6 PROGRAMAS LINEARES

6.1 RECAPITULANDO

Com o que estudamos até agora, já somos capazes de criar programas lineares, isto é, que não envolvem uma única sequência de passos, sem tomadas de decisão. Considerei prudente a inserção deste capítulo para revisarmos os conceitos estudados até então e apresentar algumas ferramentas novas, que serão úteis para os capítulos seguintes

Para começar, considere a seguinte proposta:

“Construa um programa que receba um nome e dois valores reais do usuário, e informe, chamando o usuário por seu nome, o resultado da multiplicação e da soma entre esses valores.”

Podemos escrever um algoritmo que atenda ao pedido com tudo que estudamos até aqui. Uma possibilidade de solução seria:

Algorithim Resultado da multiplicação e da soma

Input: nome e dois números reais.

Output: resultado da soma e do produto entre eles, de forma nominal.

► Declaração das variáveis utilizadas

declare:

a, b, soma, multiplicacao: **integer**

nome: **string**

► Recebimento dos dados de entrada

read a,b

read nome

►Processamento dos dados

$soma \leftarrow a + b$

$multiplicacao \leftarrow a \times b$

►Escrita dos dados de saída

write “*Bom dia,* ”, nome, “ *a soma dos valores informados é* ”, soma, “*e a multiplicação é* ”, multiplicacao

end Algorithim

6.2 PROGRAMAÇÃO POR DIAGRAMA DE BLOCOS

Até agora, programamos com base em um texto, e é assim que permaneceremos. No entanto, para fins didáticos, pode ser interessante representar nossos programas em blocos e linhas, de forma que o fluxo de execução possa ser mais facilmente seguido.

Além disso, é hora de apresentar uma nova ferramenta, que permite executar o código escrito, mesmo que ele não seja feito para nenhuma linguagem de programação específica.

Trata-se do Flowgorithm, que pode ser baixado em http://www.flowgorithm.org. Ele permite que a programação seja realizada em blocos, e será usado a partir de agora para ilustrar os comandos condicionais e de repetição.

Essa é a interface inicial do programa. Note que ela contém dois blocos, que representam, justamente, os pontos de saída e de fim da execução do algoritmo.
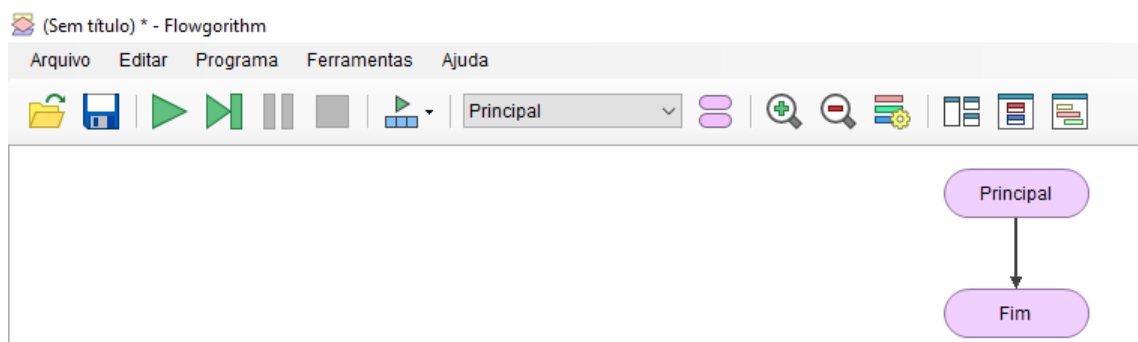
Figura 3: interface do Flowgorithm

Com o que conhecemos, já podemos representar nosso algoritmo no Flowgorithm. O primeiro passo para tanto é declararmos as variáveis que serão utilizadas. Assim, temos:
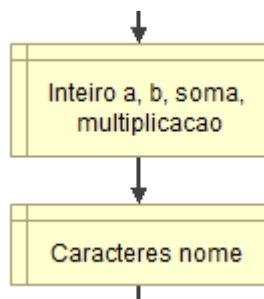
Figura 4: declaração de variáveis

O próximo passo é o recebimento dos valores inseridos pelo usuário nas variáveis criadas. Assim, temos:
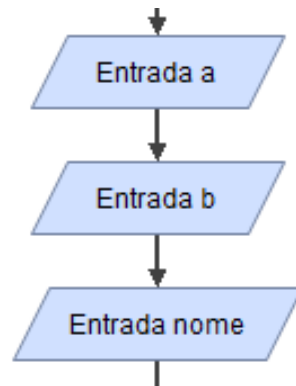
Figura 5: Recebimento de valores do usuário.

Em seguida, as variáveis soma e multiplicacao recebem seus respectivos valores:
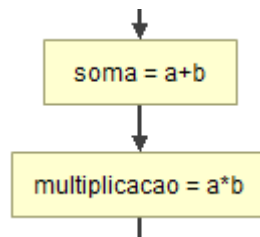
Figura 6: operações realizadas com as variáveis.

Feitos os cálculos, é hora da impressão dos resultados:
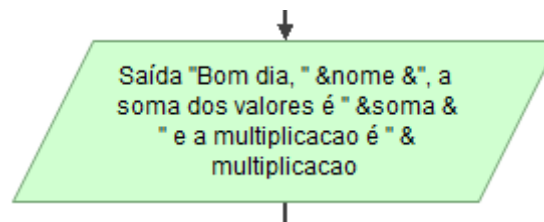
Figura 7: escrita dos valores de saída.

Assim, o resultado final do programa é o seguinte diagrama de blocos:
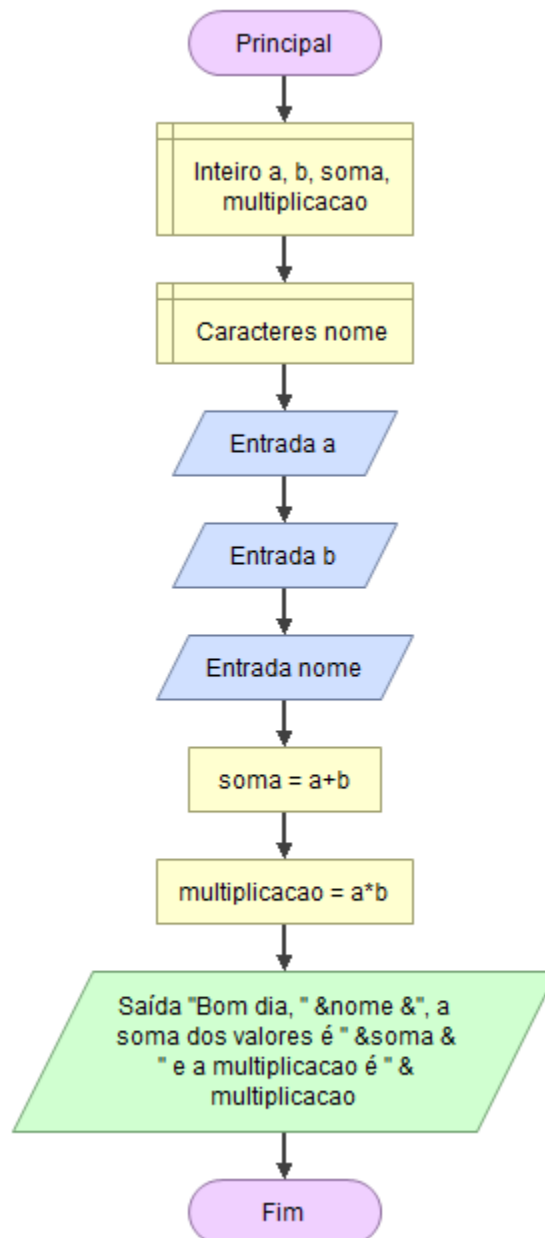
Figura 8: programa final, escrito na forma de diagrama de blocos.

O Flowgorithm ainda permite que você veja o código fonte escrito em alguns padrões de pseudocódigo (que são ligeiramente diferentes do padrão aqui adotado) ou nas principais linguagens de programação existentes.

Observando o diagrama de blocos, não é difícil perceber por que chamamos tal programa de “linear”. Nos próximos capítulos, aprenderemos a escrever programas não lineares, com estruturas que envolvem a tomada de decisão.

7 EXPRESSÕES RELACIONAIS E LÓGICAS

Antes de abordar esse tópico, teremos que, antes, abordar alguns conceitos da lógica proposicional de forma superficial.

Definimos uma **proposição** como uma sentença declarativa, que pode assumir valores-verdade V (indicando que a sentença é verdadeira) ou F (indicando que a sentença é falsa).

O que isso quer dizer? Em termos leigos, toda afirmação que pode ser verdadeira ou falsa é uma preposição. Por exemplo, “2 é maior que 3”, “a palavra “fabrica” tem 7 letras”, “hoje faz sol” são preposições, pois podem ser verdadeiras ou falsas.

Na programação, é comum que uma sequência de passos seja executada somente se uma proposição for verdadeira ou for falsa. O que veremos neste capítulo é, justamente, como escrever uma proposição.

Na maioria dos casos, só faz sentido utilizar uma proposição como parâmetro para executar ou não uma ação se seu valor verdade for variável (ou seja, pode ser V ou F). Logo, nada mais sensato do que estruturarmos nossas proposições com base em uma relação entre uma ou mais variáveis, a qual pode ser, portanto, verdadeira ou falsa.

7.1 OPERADORES RELACIONAIS

Começaremos com as variáveis numéricas. Você pode formular uma proposição entre duas variáveis numéricas, ou entre uma variável numérica e um número com os seguintes operadores. Os dois primeiros servem, também, para comparar strings.

Operador	Função
=	Igual
≠	Diferente
>	Estritamente maior
<	Estritamente menor
≥	Maior ou igual
≤	Menor ou igual

Tabela 4: operadores relacionais.

Com tais operadores, podemos formular proposições e determinar seus valores-verdade. A seguir, há alguns exemplos de expressões em algoritmo, seguidas de seus respectivos valores-verdade.

Algorithim Valores verdade

Input: não há.

Output: não há.

declare:

a, b: **integer**

r,s: **boolean**

$2 + 2 = 4$ ▶ V

$2 + 2 > 4$ ▶ F

$2 + 2 \geq 4$ ▶ V

$a \leftarrow 2$

$b \leftarrow 5$

$2 + 2 = 4$ ▶ V

$a + b = 7$ ▶ V

"batata" = "batata" ▶ V

compLiteral("batata") = compLiteral("teste") ▶ F

$r \leftarrow$ **true**

$s \leftarrow$ **false**

$s = r$ ▶ F

$s \neq r$ ▶ V

7.2 Operadores Lógicos

Com o que sabemos até agora, somos capazes de criar uma proposição comparando, por meio dos operadores relacionais, duas variáveis ou constantes, de forma que cada proposição retorna um valor-verdade.

Por meio dos operadores lógicos, podemos negar uma proposição ou combinar duas ou mais proposições, o que origina uma nova proposição e que, por sua vez, também retorna um valor-verdade

Os operadores lógicos e suas respectivas notações são:

Operador	Função
not	Não
and	E
or	Ou

Tabela 5: operadores lógicos

Tais operadores funcionam da mesma forma que na lógica proposicional. O operador not é responsável por inverter o resultado da proposição que o recebe. Por exemplo, not(2+2 = 4) retorna valor-verdade F, e not(2+2=5) retorna valor-verdade V.

A tabela-verdade para o operador not é a seguinte:

Proposição	Retorno
not(V)	F
not(F)	V

Tabela 6: tabela-verdade para o operador not.

Já o operador and só retorna valor-verdade V se todas as proposições comparadas forem verdadeiras, retornando F caso alguma das proposições comparadas for falsa. Assim, sua tabela-verdade é:

Proposição	Retorno
V and V	V
V and F	F
F and V	F
F and F	F

Tabela 7: tabela-verdade para o operador and

Por sua vez, o operador or só retorna valor-verdade F se todas as proposições forem falsas, e retorna valor-verdade V caso alguma proposição seja verdadeira. Logo, sua tabela-verdade é:

Proposição	Retorno
V or V	V
V or F	V
F or V	V
F or F	F

Tabela 8: tabela-verdade para o operador or.

Com tais operadores, podemos formular proposições e determinar seus valores-verdade. A seguir, há alguns exemplos de expressões em algoritmo, seguidas de seus respectivos valores-verdade.

Algorithim Valores verdade

Input: não há.

Output: não há.

declare:

a, b: **integer**

r,s: **boolean**

2 + 2 = 4 **and** 2 + 2 = 5 ► F

2 + 2 > 4 **or** 2 + 2 = 4 ► V

$a \leftarrow 2$

$b \leftarrow 5$

$a = b$ **or** $a \neq b$ ► V

$a = b$ **and** $a \neq b$ ► V

not $(a = b)$ **or** **not**$(a \neq b)$ ► V

8 Comandos condicionais

Até agora, nos referíamos aos programas não lineares como aqueles que envolviam uma tomada de decisão. O que exatamente isso significa?

Imagine um algoritmo que atenda à seguinte proposta:

"Construa um programa que receba dois valores e diga se o produto entre eles é positivo ou negativo"

Com o que sabemos até aqui, não teríamos problemas em construir o algoritmo até a parte de impressão dos resultados. Ele seria:

```
Algorithim Sinal do produto de dois valores

Input: dois valores inteiros.

Output: se o produto entre eles é positivo ou negativo

declare:

        a, b, produto: integer

read a,b

produto ← a × b
```

Entretanto, "travamos" ao escrever os resultados. Como dizer para o algoritmo realizar uma dada ação apenas caso uma condição seja satisfeita?

8.1 Estrutura if

Para tanto, lembre-se das proposições que estudados no capítulo anterior. A afirmação "um número multiplicado por outro resulta em um número positivo" é uma proposição, que pode assumir valores-verdade V ou F.

Logo, podemos condicionar uma ação para ocorrer apenas caso uma proposição assuma valor verdade V (ou F). Para isso, utilizamos a estrutura if, da seguinte forma:

```
if produto ≥ 0 then

        write "O produto entre os números é positivo"

else

        write "O produto entre os números é negativo"

end if
```

Assim, a proposição ($produto \geq 0$) passa a condicionar a ação. Se ela tiver valor-verdade V, é executado o conjunto de passos entre o then e o else. Caso ela tiver valor-verdade F, o conjunto de passos a ser executado passa a ser aquele entre o else e o end if.

Assim, o algoritmo final fica:

```
Algorithm Sinal do produto de dois valores

Input: dois valores inteiros.

Output: se o produto entre eles é positivo ou negativo

declare:

        a, b, produto: integer

read a,b

produto ← a × b


if produto ≥ 0 then

        write "O produto entre os números é positivo"

else

        write "O produto entre os números é negativo"

end if


end Algorithm
```

Podemos ainda reescrever o código na forma de diagrama de blocos, obtendo:
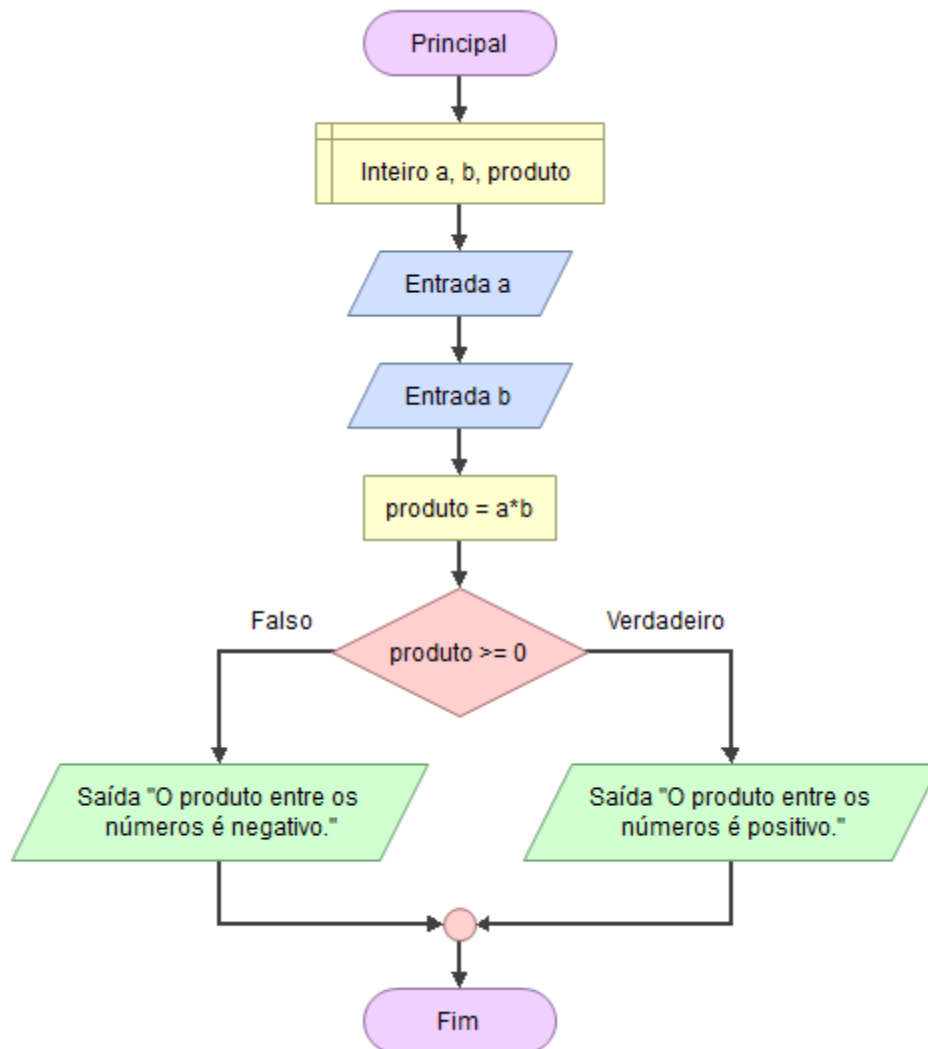
Figura 9: diagrama de blocos do pseudocódigo estudado.

Note que acima utilizamos apenas uma proposição para analisar o valor-verdade. Entretanto, poderíamos ter utilizado qualquer tipo de proposição, ou ainda conjuntos de proposições, ligadas por operadores or ou and.

Outro detalhe importante a considerarmos é que, talvez, não haja nenhuma ação a ser executada caso o valor-verdade da proposição em questão seja F. Nesse caso, podemos suprimir a sintaxe else, e terminar o laço com o end if.

Muitas vezes, encontramos situações nas quais existem mais de uma opção, mutuamente exclusivas, para uma tomada de decisão. Por exemplo, considere que em um programa haja um menu, que funciona da seguinte forma:

“Uma variável recebe um número digitado pelo usuário, que pode ser 1,2, ou 3. Se o resultado for 1, o programa exibe a mensagem “Palmeiras”, se for 2, exibe “Internacional” e, se for 3, exibe “Flamengo”. Se não for nenhum desses valores, exibe “Não encontrado”.”

Podemos configurar tal menu a partir de comandos if aninhandos, ou seja, um comando if dentro de outro comando if, até que todas as possibilidades sejam exauridas. Veja, abaixo, como a implementação seria em pseudocódigo:

```
Algorithm Três primeiros colocados no campeonato brasileiro no dia 08/10/2018

Input: número de 1 a 3.

Output: time correspondente à colocação inserida.

declare:

        menu: integer

read menu

if menu = 1 then

        write "Palmeiras"

else

        if menu = 2 then

                write "Internacional"

        else

                if menu = 3 then

                        write "Flamengo"

                else

                        write "Não encontrado"

                end if

        end if

end if

end Algorithm
```

Essa implementação também pode ser reproduzida em diagrama de blocos, conforme mostrado abaixo.
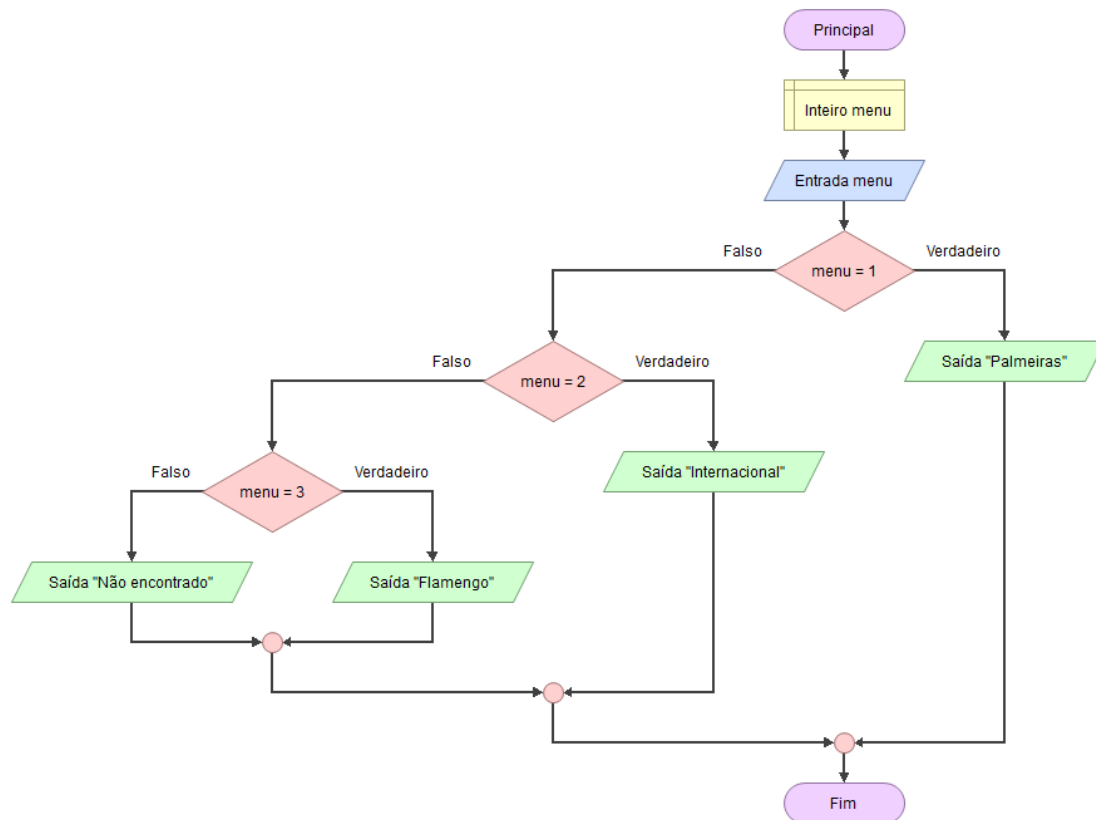
Figura 10: algoritmo do pseudocódigo estudado.

8.2 ESTRUTURA ELSE IF

Você deve ter notado que a técnica de comandos if aninhados não funciona bem quando temos um número muito grande de possibilidades, uma vez que dificulta a legibilidade do código. Justamente por essa razão existem as estruturas else if e switch.

Primeiro, estudaremos a estrutura else if. Ao invés dessa estrutura trabalhar com o valor-verdade de uma proposição, ela checa o valor verdade de várias proposições, até encontrar uma que retorne valor-verdade V. Nesse ponto, ela executa o código presente nessa proposição.

Observe, abaixo, o mesmo algoritmo escrito utilizando uma estrutura else if no lugar dos comandos if aninhados.

Algorithm Três primeiros colocados no campeonato brasileiro no dia 08/10/2018

Input: número de 1 a 3.

Output: time correspondente à colocação inserida.

declare:

```
    menu: integer
read menu
if menu = 1 then
    write "Palmeiras"
else if menu = 2 then
    write "Internacional"
else if menu = 3 then
    write "Flamengo"
else
    write "Não encontrado"
end if
end Algorithm
```

8.3 ESTRUTURA SWITCH

Essa estrutura funciona de forma muito semelhante à estrutura else if, com a diferença de que, ao passo que a primeira checa o valor-verdade de proposições que podem ser totalmente diferentes entre si, a switch apenas compara o retorno (numérico ou de texto) de uma expressão com outros valores, também numéricos ou de texto.

Observe, abaixo, o mesmo código escrito com uma estrutura switch.

```
Output: time correspondente à colocação inserida.
declare:
    menu: integer
read menu
switch menu of
    case 1 do
        write "Palmeiras"
    end case
    case 2 do
```

```
            write "Internacional"
        end case
        case 3 do
            write "Flamengo"
        end case
        otherwise do
            write "Não encontrado"
        end otherwise
end switch
end Algorithm
```

Como você pode perceber, a sintaxe otherwise faz o papel do else na estrutura if.

8.4 COMO LIDAR COM CONDIÇÕES NÃO-MUTUAMENTE EXCLUSIVAS?

Até agora, vimos que as estruturas else if e switch funcionam muito bem para condições mutuamente exclusivas (isto é, aquelas em que duas condições nunca ocorrem simultaneamente) e que as estruturas if aninhadas podem ser usadas sem pudor nessas condições.

Considere, agora, o seguinte problema:

"Faça um programa que informe se um número é múltiplo de 2, de 3 ou de ambos."

Nesse caso, usar qualquer uma das estruturas acima sem se importar com a ordem em que elas aparecem poderia causar problemas. Por exemplo, se optássemos por verificar se um número é múltiplo de 2 antes de verificar se ele é múltiplo de 6, correríamos o risco de não dizer que 12 é múltiplo de 6.

Para situações do tipo, não há um processo único e totalmente funcional, mas sim recomendações que você pode seguir.

A principal delas é usar comandos if aninhados, colocando as possibilidades menos gerais dentro de outras, mais gerais.

Por exemplo, podemos fazer:

```
Algorithm Múltiplo de 2,3 ou 6
```

```
Input: número inteiro.
Output: se o número inserido é múltiplo de 2,3 ou 6
declare:
        numero: integer
read numero
if resto(numero, 6) = 0 then
        write "O número é múltiplo de 6"
        write "O número é múltiplo de 3"
        write "O número é múltiplo de 2"
else
        if resto(numero, 3) = 0 then
                write "O número é múltiplo de 3"
        else if resto(numero, 2) = 0 then
                write "O numero é múltiplo de 2"
        end if
end if
end Algorithm
```

A qual é uma solução funcional, uma vez que, se um número for múltiplo de 6, ele certamente será múltiplo de 2 ou de 3 e, portanto, essa verificação não é necessária.

Além disso, a condição mais geral (o número ser múltiplo de 6) é verificada primeiro e, uma vez eliminada, ela faz com que as opções seguintes se tornem mutuamente exclusivas, permitindo a utilização de uma estrutura else if.

9 COMANDOS DE REPETIÇÃO

Imagine, agora, que queiramos construir um programa que se adeque a seguinte proposta.

"Construa um programa que imprima os números de 1 até 10".

Usando a estratégia de refinamentos sucessivos, podemos construir uma primeira versão de nosso algoritmo da seguinte forma:

```
Definir um contador i.
Assinalar i = 1.
Se i ≤ 10:
    Imprimir i.
    Acrescer i em 1 unidade.
    Repetir a terceira linha.
Se não:
    Encerrar o algoritmo.
```

Nesse ponto, já temos um problema. Como poderíamos indicar ao algoritmo o ato de "repetir"? É aqui que entra o tema dessa unidade, os comandos de repetição.

Para exemplificarmos os laços, iremos extrapolar o diagrama de blocos, inserindo uma linha que retorna ao bloco original – algo que não é permitido nos algoritmos.

9.1 LAÇO FOR

O laço for tem a seguinte a estrutura, considerando um diagrama de blocos extrapolado.
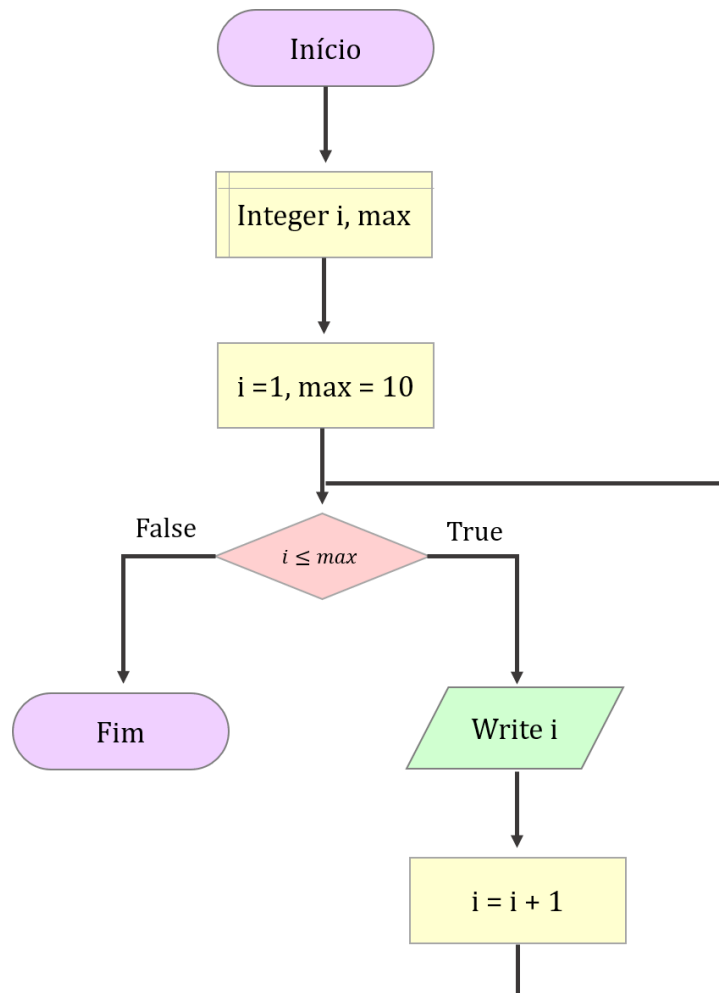
Figura 11: representação de um laço for em diagrama de blocos, abusando da sintaxe.

Porém, ainda há um problema: como fazer essa operação de retorno em um programa escrito?

É justamente para contornar tal problema que surge o laço for. Veja, abaixo, um exemplo da mesma implementação.

```
for i ← 1 to 10 step 1 do
    ▶ Comando a ser executado em cada vez
end for
```

Aqui, é prudente analisarmos o funcionamento do laço for. Em primeiro lugar, i corresponde a uma variável de controle, que pode assumir qualquer nome. Ela é utilizada como contador do laço.

O valor em azul, colocado após o step, é o “passo” do laço, e corresponde ao valor que é acrescido à variável de controle em cada repetição.

O valor em vermelho corresponde ao valor inicial da variável de controle e, como você pode notar no diagrama de blocos, ele será o valor que tal variável irá assumir na primeira execução.

Por sua vez, o valor em verde corresponde ao valor máximo que a variável de controle irá assumir antes que o laço seja interrompido.

Só para exemplificar, suponha a execução do código abaixo:

```
Algorithm: contar até dez
Input: não há.
Output: não há.

for i ← 1 to 10 step 1 do
    write i
end for
end Algorithm
```

O resultado da compilação será:

```
1
2
3
4
5
6
7
8
9
10
```

Ou seja, ele irá executar operação para $i = 1$, $i = 10$ e todos os valores entre eles. Ao final, a variável i receberá o valor 11, mas o laço não será executado, por a expressão $11 \leq 10$ possui valor-verdade F.

Isso pode parecer óbvio, mas é um dos erros mais comuns nessa parte do conteúdo. Muitos estudantes, ao realizar um laço for, acabam por inserir um valor imediatamente inferior ou imediatamente superior em uma das posições, provocando grandes erros.

Além disso, existem mais algumas diretrizes sobre o funcionamento do laço for que devem ser levados em consideração:

1. Caso o valor inicial seja maior que o valor final, os comandos internos não serão executados nenhuma vez e o valor da variável de controle será simplesmente igual ao valor de início (não haverá incremento, pois não houve execução);

2. Se o valor de início for igual ao valor de fim, os comandos serão executados uma única vez e a variável de controle terminará com valor incrementado de 1;

3. Em pseudocódigo, não é necessário declarar previamente a variável de controle, por mais que o Flowgorithm e algumas linguagens de programação exijam isso;

4. Não se pode alterar o valor da variável de controle dentro do laço;

5. Caso os valores de início e de fim sejam formados por expressões que usam variáveis, estas expressões serão avaliadas somente uma vez antes das repetições começarem; portanto, se houver modificação dos valores das variáveis, o número de repetições estimado inicialmente não será alterado.

Por fim, esse laço é ilustrado, em diagrama de blocos, da seguinte forma:
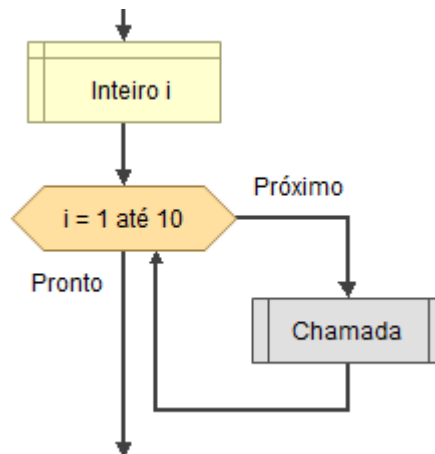
Figura 12: laço for em diagrama de blocos.

9.2 LAÇO WHILE

O comando for pode parecer basta útil, mas há situações em que ele simplesmente não é suficiente. Por exemplo, imagine uma solução para o seguinte problema:

“Crie um programa que recebe uma lista de nomes e os imprima na tela seguidos de uma saudação, até que o nome inserido seja “fim”. Nesse caso, encerre o programa”.

Aqui, ainda utilizando a técnica de refinamentos sucessivos, uma possível solução seria:

```
Declarar a variável nome
Solicitar a inserção de um nome
Se nome ≠ “fim”
        Imprimir o nome
        Solicitar a inserção de um nome
        Voltar para a terceira linha
Se nome = “fim”
        Encerrar o algoritmo
```

Dessa vez, sabemos como indicar ao algoritmo o ato de “repetir”, mas não sabemos quantas vezes a mesma ação (de solicitar o nome) deverá ser realizada. Uma forma de esquematizar esse algoritmo em diagrama de blocos seria:
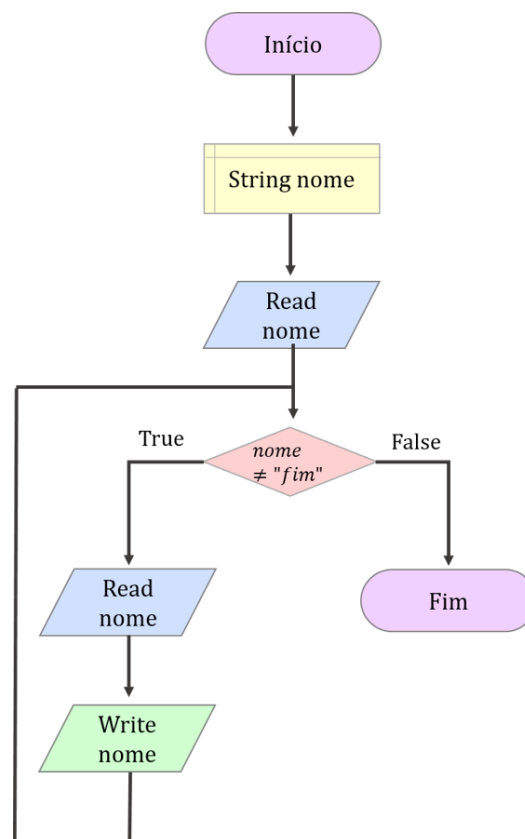
Figura 13: representação de um laço while em diagrama de blocos, abusando da sintaxe.

O problema da situação anterior permanece: como fazer com que o algoritmo repita o passo anterior, dessa vez por um número indeterminado de vezes?

A solução encontra-se no laço while. Observe, abaixo, uma implementação de tal estrutura.

while nome ≠ fim **do**

▶ Comando a ser executado enquanto a expressão nome ≠ fim tiver valor-verdade V

end while

Assim, entre as palavras while e do deve vir uma proposição lógica. Se ela tiver valor-verdade V, os comandos presentes dentro do laço serão executados, e o laço retornará para o ponto inicial, verificando, mais uma vez, o valor-verdade da expressão e reiniciando o ciclo.

Quando, em algum momento, o valor-verdade da expressão for F, os comandos dentro do laço não serão executados, e ele terminará.

Em diagrama de blocos, o laço while pode ser representado da seguinte forma:
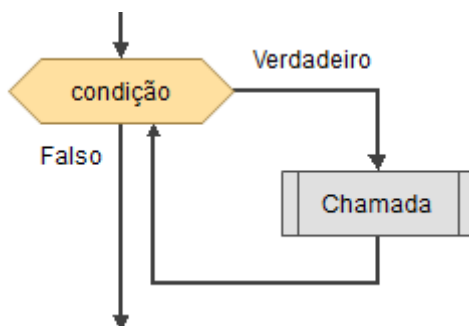
Figura 14: laço while em diagrama de blocos.

9.3 LAÇO REPEAT

Por fim, há outra forma de solucionar o mesmo problema. Ao invés de dizermos ao programa "faça determinada ação enquanto tal proposição for verdadeira", podemos também dizer "faça determinada ação até que tal proposição seja falsa".

Nesse caso, utilizamos o laço repeat. O diagrama de blocos por abuso de notação seria praticamente o mesmo, com a diferença que condiciona seu fim, é agora, negada. Veja a seguir.
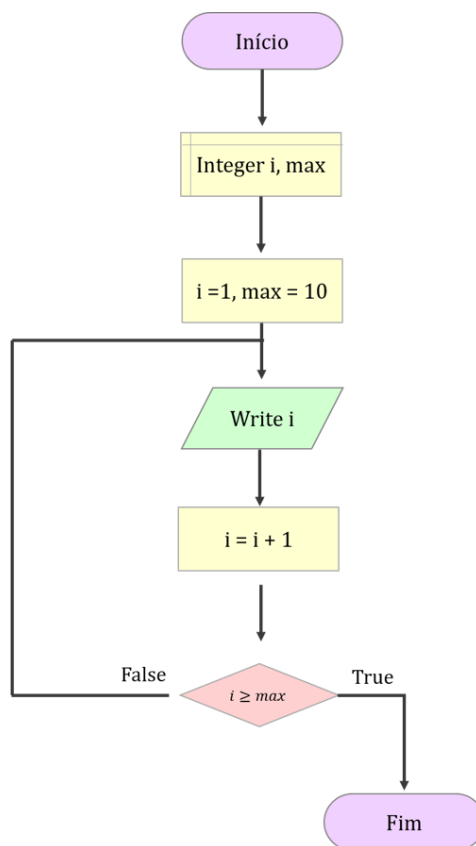
Figura 15: representação de um laço repeat em diagrama de blocos, abusando da sintaxe.

A sintaxe em algoritmo também muda. Veja, abaixo, o mesmo problema da seção anterior implementado com um laço repeat.

repeat

▶ Comando a ser executado enquanto a expressão nome = fim tiver valor-verdade F

until nome = fim

Assim, o comando entre o repeat e o until será executado até que a expressão após o until tenha valor-verdade V.

Ao lidar com esses dois laços, existem três cuidados que devem ser tomados.

Em primeiro lugar, é extremamente comum, para estudantes iniciados, escrever programas que produzem um loop infinito quando executados, o que o ocorre por que a condição que deve encerrar o loop nunca ocorre. Para evitar que isso aconteça, deve-se revisar o código e “testá-lo” mentalmente, visando encontrar situações nas quais ele não terá fim.

Em segundo lugar, deve-se lembrar que as expressões que condicionam o fim do laço em while e repeat são diferentes umas das outras (afinal de contas,

para o primeiro caso, o laço continua se a expressão tiver valor-verdade V e, para o segundo caso, se a expressão tiver valor-verdade F).

Por fim, para comandos while, é possível que um conjunto de comandos não seja executado nenhuma vez (caso a condicional nunca se verifique). Já para comandos repeat, esse conjunto sempre será executado ao menos uma única vez, uma vez que a verificação da condição só ocorre depois da execução do conjunto de comandos.

O Flowgorithm não possui exatamente uma implementação da função repeat, mas possui uma estrutura que tem uma singela diferença: ela continua caso o valor-verdade da expressão condicional seja V, e não F, como no laço que estudamos.
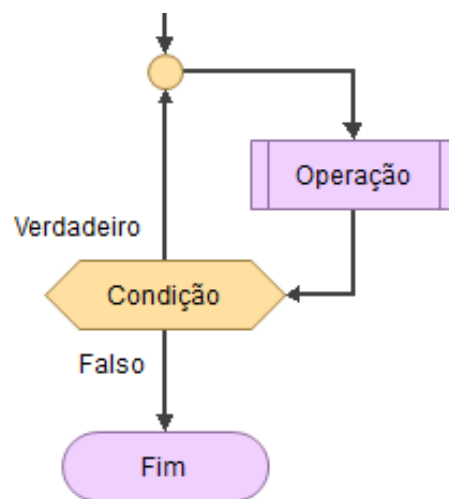
Figura 16: função do Flowgorithm que funciona semelhante ao laço repeat.

10 PROGRAMAS NÃO-LINEARES E TESTE DE MESA

10.1 O QUE JÁ SOMOS CAPAZES DE FAZER

Com o que vimos até aqui, já somos perfeitamente capazes de codificar soluções para a maioria dos problemas que um estudante encontrará no início de seu curso de programação.

Nessa seção, proponho a solução de um exercício, que resume tudo que estudamos até aqui em um único programa:

"Faça um programa que, para cada aluno, receba seu nome, o valor de suas n notas (com n informado pelo usuário) e informe se ele foi ou não aprovado (o aluno é aprovado caso tenha média maior ou igual a 6) e sua média. Quando o nome do aluno informado for "fim", o programa deverá encerrar."

A seguir, apresento a solução comentada para o problema.

```
1
2   ► O cabeçalho é feito conforme os padrões que já estudamos
3   Algorithm: alunos, suas médias e aprovações
4   Input: nome de alguns alunos, quantidade de notas de cada aluno e suas notas
5   Output: a média final do aluno e se ele foi ou não aprovado
6
7   ► Declaração das variáveis utilizadas
8   declare:
9           nome: string
10          nota: real
11          media: real
12          qtd_notas: integer
13  ► Valores padrão para variáveis
14  ► Permite que a variável nome seja comparada com "fim", logo abaixo
15          nome ← "genérico"
16  ► Execução do código
17  while nome ≠ "fim" do
18          read nome
19          ► Essa condição é necessária, já que, caso o nome for igual a "fim", não se deve realizar
20  a leitura das notas
```

```
21          if nome ≠ "fim" then
22                media ← 0
23                read qtd_notas
24                for i ← 1 to qtd_notas step 1 do
25                      read nota
26                      media ← media + nota
27                end for
28                media ← media / qtd_notas
29                if media ≥ 6 then
30                      write "Aluno aprovado com media", media
31                else
32                      write "Aluno reprovado com media", media
33                end if
34          end if
35    end while
36    end Algorithm
```

Se quisermos ver nosso código funcionando, podemos construí-lo no Flowgorithm, cujo resultado encontra-se na próxima parte.
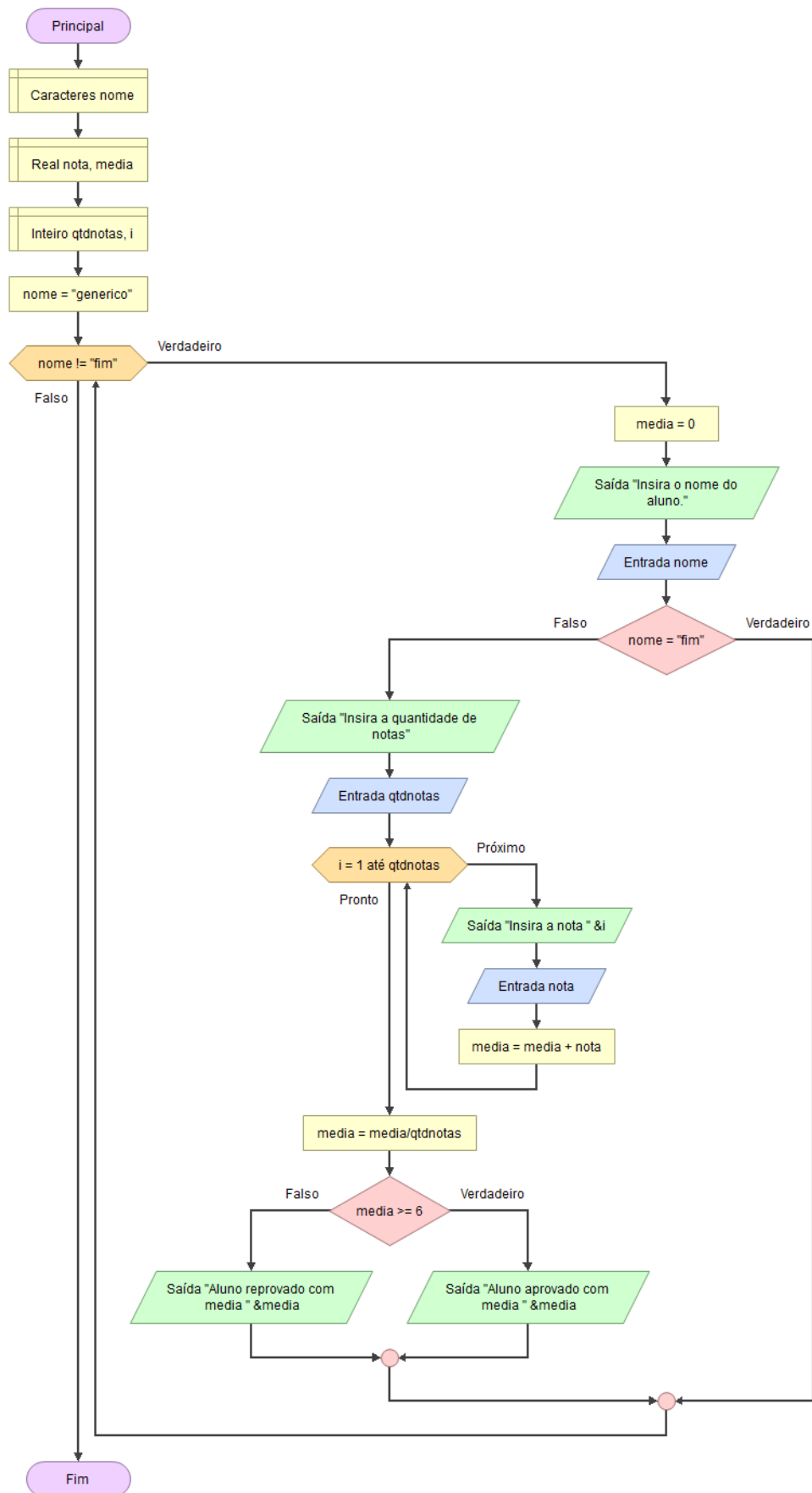
Figura 17: algoritmo criado em diagrama de blocos

10.2 Solução adequada

Sabemos como construir um algoritmo, mas como saber se ele se adequa ao problema proposto?

Um algoritmo se adequará ao problema proposto se todos seus valores de saída forem compatíveis com seus valores de entrada, para todos os casos possíveis para o problema.

Em termos mais leigos, isso significa dizer que, se o output de um algoritmo estiver correto para todos os valores de entrada que o algoritmo pode receber em uma situação real, então o algoritmo se adequa ao problema.

Em nosso exemplo, é preciso que o output seja correto para a entrada de qualquer nome composto por caracteres, para a entrada de qualquer quantidade natural de notas e para a entrada de qualquer valor real positivo para cada nota. Se o algoritmo retornar o resultado esperado nessas condições, então ele se adequa ao problema.

Essa definição pode parecer óbvia, mas ela nos permite concluir que um algoritmo não precisa lidar com as exceções (ou seja, valores de entrada que fogem daqueles reais) para se adequar.

Ou seja, sabemos que o algoritmo irá falhar caso a nota de um aluno seja inserida como texto, mas como esse é um caso que foge de uma situação real, ele não impede que o algoritmo seja considerado adequado.

10.3 Teste de mesa

Uma das formas de identificar possíveis erros é através da execução de um teste de mesa. Ele não garante que o algoritmo se adeque a todos os casos possíveis, mas pode ajudar o programador a identificar casos em que ele não se adeque.

Para realizar um teste de mesa, deve-se percorrer o algoritmo linha a linha, realizar as entradas de valores necessários e manter controle dos valores que cada variável assume após cada passo sendo executado.

Existem diversas formas de realizar o teste de mesa, e a escolha da melhor forma para fazê-lo fica a critério do programador.

Aqui, sugerimos a construção de uma tabela, com as colunas “Linha”, “Input” e Output”, assim como colunas subsequentes com o nome de cada uma das variáveis utilizadas no teste.

A primeira coluna corresponde a linha de execução do código, ao passo que a segunda e terceira coluna correspondem, respectivamente, a valores inseridos e retornados pelo código.

Por fim, as demais colunas contêm os valores assumidos pelas variáveis em cada linha do programa. Linhas que não alteram nenhuma variável do código (comentários, cabeçalhos, rodapés, tomadas de decisão) podem ser desprezadas.

Abaixo, há um exemplo de teste de mesa para o algoritmo que criamos.

Linha	**Input**	**Output**	**Nome**	**nota**	**media**	**qtd_notas**	**I**
15			Genérico				
18	João		João				
22			João		0		
23	3		João		0	3	
24			João		0	3	1
25	8		João	8	0	3	1
26			João	8	8	3	1
24			João	8	8	3	2
25	9		João	9	8	3	2
26			João	9	17	3	2
24			João	9	17	3	3
25	6		João	6	17	3	3
26			João	6	23	3	3
28			João	6	7,66	3	4
29		Aluno aprovado com média 7,66	João	6	7,66	3	4
18	Maria		Maria	6	7,66	3	4
22			Maria	6	0	3	4
23	2		Maria	6	0	2	4
24			Maria	6	0	2	1
25	4		Maria	4	0	2	1
26			Maria	4	4	2	1
24			Maria	4	4	2	2
25	5		Maria	5	4	2	2
26			Maria	5	9	2	3
28			Maria	5	4,5	2	3
29		Aluno reprovado com média 4,50	Maria	5	4,5	2	3
18	Fim		Fim	5	4,5	2	3

Vale lembrar que o teste de mesa não necessariamente envolve o rigor que utilizamos acima, e pode até mesmo ser executado mentalmente pelo programador. Além disso, essa tarefa pode tornar-se muito mais fácil com o uso do Flowgorithm, por meio do monitoramento de variáveis.
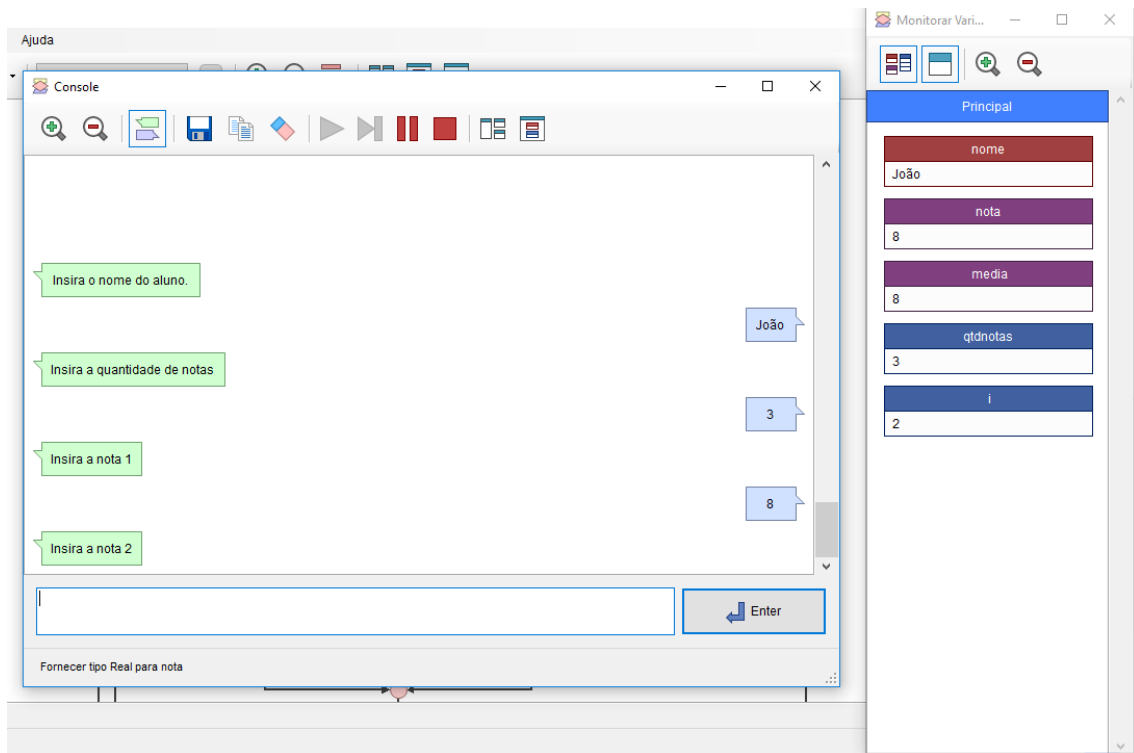
Figura 18: teste de mesa utilizando o Flowgoritm

11 ARRANJOS

11.1 ARRANJOS UNIDIMENSIONAIS

Considere, agora, o seguinte problema:

"Construa um programa que receba as três notas de um aluno em uma disciplina. Em seguida, o programa deve imprimir um boletim, contendo cada uma das notas e a média final."

Uma possibilidade de solução seria utilizar três variáveis diferentes, de nomes nota1, nota2 e nota3, e prosseguir com a solução do problema, que pode ser feita de forma linear. Tal solução compreenderia o seguinte algoritmo:

```
Algorithm: boletim
Input: notas de um aluno em 3 provas.
Output: boletim contendo nota de cada prova e sua média final


declare:
        nota1, nota2,nota3: real
        media: real
read nota1, nota2, nota3
media ← (nota1 + nota2 + nota3) / 3
write nota1, nota2, nota3
write media
end Algorithm
```

Entretanto, se a quantidade de notas aumentasse para 5 notas, um programador deveria alterar a declaração das variáveis, todas as estruturas de entrada e todas as estruturas de saída, além das duas operações que compõe o cálculo da média final.

Tal problema poderia ser contornado se fôssemos capazes de armazenar todas as n notas em uma mesma variável. É nesse contexto que surge o conceito de arranjos (ou vetores) de uma dimensão.

Um vetor de uma dimensão pode ser ilustrado da seguinte forma:
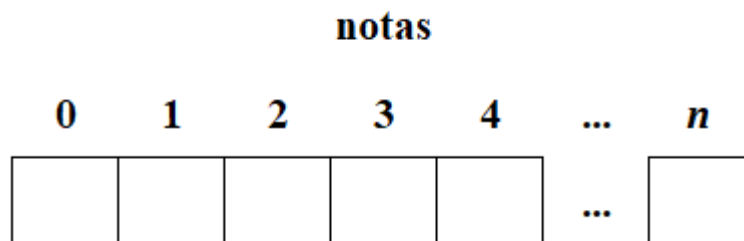
Figura 19: vetor de uma dimensão notas, com $n + 1$ elementos.

Em pseudocódigo, um vetor de uma dimensão e tamanho 3 pode ser declarado da seguinte forma:

```
declare:
    notas[3]: inteiro
▸ A estrutura utilizada para a declaração sempre será
nomeArranjo[elementos]:tipoArranjo
```

Como você pode perceber, um arranjo pode conter variáveis de qualquer tipo, desde de que todas elas sejam do mesmo tipo. Assim, podemos ter um arranjo de strings, de inteiros, de números reais, de ponteiros e de estruturas – esses dois tipos de dados serão abordados adiante.

Aqui, há um detalhe importante a ser observado. Em todo vetor, o primeiro elemento sempre possui o índice 0 e, portanto, um vetor de n elementos deve ser declarado com o valor $n - 1$ entre colchetes.

Quando desejamos acessar ou operar algum elemento do vetor, basta o especificarmos utilizando os colchetes. O valor entre colchetes é denominado índice do elemento.

Assim, podemos, por exemplo atribuir o valor da terceira nota de um aluno a uma variável chamada temp por meio do seguinte código:

```
temp ← notas[2]
```

Ou, ainda, adicionar 2 pontos a quarta nota de um aluno por meio do seguinte código:

```
notas[2]← notas[2] + 2
```

Em suma, elementos de um vetor funcionam como qualquer variável, estando sujeitos às mesmas operações e regras que já estudamos até aqui.

Por fim, é interessante utilizar um laço for toda vez que for necessário percorrer todos os elementos de um arranjo unidimensional. Veja, abaixo, o exemplo que iniciou esta seção escrito utilizando um vetor.

```
Algorithm: boletim
Input: notas de um aluno em 3 provas.
```

```
Output: boletim contendo nota de cada prova e sua média final
► Declaração das variáveis utilizadas
declare:
        notas[3]: real
        media: real
► Leitura das notas
        media ← 0
► Leitura das notas
for i → 0 to 2 step 1 do
        read notas[i]
        media ← media + notas[i]
end for
► Determinação da média
media ← media / 3
► Escrita dos resultados
for i → 0 to 2 step 1 do
        write notas[i]
end for
write media
end Algorithm
```

11.2 ARRANJOS BIDIMENSIONAIS

Considere, agora, o seguinte problema:

“Construa um programa que receba os elementos de uma matriz com n *linhas e* m *colunas, os quais podem ser, no máximo, 50. Em seguida, o programa deve exibir a matriz na tela.”*

Aqui, não utilizar arranjos já se torna inviável, uma vez que seriam necessárias $50 \times 50 = 2500$ variáveis, uma para cada elemento da matriz.

Assim, a solução passa a envolver o uso de um arranjo bidimensional, o qual pode ser ilustrado da seguinte forma:

Figura 20: vetor de duas dimensões matriz, com $(n + 1) \times (n + 1)$ elementos.

Em pseudocódigo, uma matriz de reais com 50 linhas e 50 colunas pode ser declarada da seguinte forma:

declare:

matriz[50][50]: **real**

▸ A estrutura utilizada para a declaração sempre será nomeArranjo[linhas][colunas]:tipoArranjo

Mais uma vez, é possível construir uma matriz com qualquer tipo de elemento, desde de que todos os elementos sejam do mesmo tipo. Além disso, a regra de começar a contar a partir do zero também vale: a primeira linha é sempre a de número zero, e a primeira coluna também é sempre a de número zero.

Analogamente, podemos operar cada elemento da matriz como operaríamos uma variável, sempre indicando a linha e a coluna com o uso de colchetes. Veja, abaixo, alguns exemplos.

temp ← matriz[3][2] ▸ atribui o elemento presente na linha 3 e coluna 2 a uma variável

write matriz [0][1] ▸ escreve na tela o elemento presente na linha 0 e coluna 1

write matriz[5][5] ▸ escreve na tela o elemento presente na linha 5 e coluna 5

write matriz[a][b] ▸ escreve na tela o elemento presente na linha de valor igual a variável a, e na coluna de valor igual a variável b

write matriz[a + 1][b − 2] ▶ escreve na tela o elemento presente na linha de valor igual a variável a + 1, e na coluna de valor igual a variável b - 2

Sempre que desejarmos realizar uma operação encadeada com a matriz (imprimir a matriz na tela, receber todos os elementos da matriz, calcular um determinante, etc), é conveniente utilizarmos dois laços for. Veja, a seguir, a implementação da solução do problema que iniciou essa seção.

```
Algorithm: matriz
Input: número de linhas e colunas de uma matriz, elementos da matriz
Output: a própria matriz inserida
► Declaração das variáveis utilizadas
declare:
        matriz[50][50]: real
        num_linhas: integer
        num_colunas: integer
► Recebimento da quantidade de linhas e colunas
        read num_linhas
        read num_colunas
► Recebimento dos elementos da matriz
        for i → 0 to num_linhas - 1 step 1 do
                for j → 0 to num_colunas - 1 step 1 do
                        read matriz[i][j]
                end for
        end for
► Impressão da matriz
        for i → 0 to num_linhas - 1 step 1 do
                for j → 0 to num_colunas - 1 step 1 do
                        write matriz[i][j]
                end for
        end for
end Algorithm
```

Antes de prosseguir, note a forma como realizamos os dois laços for: apesar da matriz possuir num_linhas linhas e num_colunas colunas, tais linhas e colunas estão identificadas por num_linhas – 1 e num_colunas – 1, respectivamente. Da

mesma forma, a primeira linha e a primeira coluna estão identificadas pelo índice 0.

Além disso, sabemos que a matriz pode ter, no máximo, 50 linhas e 50 colunas. Portanto, a declaramos com 50 linhas e 50 colunas, mas utilizamos apenas o número de linhas e o número de colunas inserido pelo usuário. Essa questão será abordada futuramente, no capítulo sobre alocação dinâmica.

11.3 ARRANJOS n-DIMENSIONAIS

Muitos autores preferem, antes de prosseguir para essa seção, abordar mais um caso particular que possui visualização gráfica, o de um arranjo tridimensional. Entretanto, acredito que o esforço necessário para visualizá-lo não é recompensador, uma vez que, muitas vezes, essa visualização sequer ajuda na solução do problema. Assim, abordaremos diretamente o procedimento para criar vetores com n dimensões.

Para declarar um vetor, basta utilizarmos a mesma sintaxe que já trabalhamos até aqui, adicionando um colchete para cada dimensão a mais que o vetor possuirá. Veja alguns exemplos:

declare:

estoque[2][4][4][3]: **integer**

▸ 1. Estoque de uma loja de calças. A primeira dimensão corresponde ao sexo (0 = masculino, 1 = feminino), a segunda dimensão corresponde ao tamanho (0,1,2,3), a terceira dimensão corresponde ao comprimento (0,1,2,3), e a quarta dimensão corresponde a cor (0 = azul, 1 = cinza, 2 = preto)

observacao[32][13][11]: **integer**

▸ 2.A primeira dimensão corresponde ao dia do ano, a segunda ao mês, e a terceira ao ano (que assume valores de 1 a 10). Aqui, optamos por desprezar os índices 0, e começar a contar a partir do 1.

pontoemR4[99][99][99][99]: **real**

▸ 3.Cada dimensão corresponde a uma coordenada de um vetor no espaço vetorial $\mathbb{R}^4$, que contém índices até 100 em cada dimensão. Se você não sabe álgebra linear, não se preocupe em entender o contexto desse exemplo.

Em todos eles, é importante que o estudante tenha um conceito em mente: os valores que devem vir dentro dos colchetes correspondem a um índice, e só existe um elemento em cada posição definida por um conjunto de colchetes. Por exemplo, no exemplo 2, chamar uma variável com observacao[18][3][5] irá retornar o valor correspondente ao dia 18 de março do ano 5.

Além disso, sempre que um elemento de um arranjo for chamado, é necessário especificar o índice de todas as suas dimensões. Por exemplo, no exemplo 2, não podemos chamar um elemento utilizando observacao[5][10].

Por fim, é interessante apresentarmos um exemplo de utilização de laços for para arranjos com mais de duas dimensões. Observe o problema a seguir, criado com base no exemplo 2.

"Um estudante deseja registrar a quantidade de peixes presentes em um aquário durante 10 anos. Crie um programa que receba essa quantidade para cada dia do mês (desprezando anos bissextos, mas considerando a quantidade de dias em cada mês) e depois, conforme inserido pelo usuário, informe o total de peixes em um dia, mês e ano específico."

Abaixo, há a solução algorítmica comentada para o problema em questão.

Algorithm: número de peixes em um aquário

Input: quantidade de peixes em cada dia, durante um período de 5 anos, dia desejado para receber essa quantidade

Output: quantidade de peixes no dia desejado

► Declaração das variáveis utilizadas

declare:

total_peixes[32][13][6]: **integer**

input_dia: **integer**

input_mes: **integer**

input_ano: **integer**

quantidade_dias[13]: **integer**

► Criação do vetor que indica quanto dias há em cada mês

quantidade_dias[1] ← 31

quantidade_dias[2] ← 28

quantidade_dias[3] ← 31

quantidade_dias[4] ← 30

quantidade_dias[5] ← 31

quantidade_dias[6] ← 30

quantidade_dias[7] ← 31

quantidade_dias[8] ← 31

quantidade_dias[9] ← 30

quantidade_dias[10] ← 31

```
        quantidade_dias[11] ← 30
        quantidade_dias[12] ← 31
► Recebimento da quantidade de peixes em cada dia
        for a → 1 to 10 step 1 do
                for m → 1 to 12 step 1 do
                        for d → 1 to quantidade_dias[m] step 1 do
                                read total_peixes[d][m][a]
                        end for
                end for
        end for
►Escrita da quantidade de peixes em um dado dia
        read input_dia, input_mes, input_ano
        write total_peixes[input_dia][input_mes][input_ano]
end Algorithm
```

12 ESTRUTURAS

Considere, mais uma vez, o seguinte problema:

"Uma estação meteorológica coleta, durante um período de 50 dias, dados relativos a umidade relativa do ar (número real), a temperatura (número real) e o tipo de nuvem (texto). Construa um programa que recebe todas essas informações e apresente a temperatura média observada nesse período."

Utilizando os conceitos do capítulo anterior, um estudante poderia construir uma solução algorítmica declarando suas variáveis na forma de três arranjos, como mostrado abaixo:

```
declare:
    umidade[50]: real
    temperatura[50]: real
    nuvem[50]: string
```

Assim, teríamos três arranjos, cada um responsável por ordenar uma variável. O dia corresponde a observação seria indicado pelo índice desses arranjos.

No entanto, o uso dessa organização pode prejudicar a legibilidade do código, além de torna-lo mais complicado do que gostaríamos. Seria interessante se pudéssemos criar um novo tipo de variável, chamada, por exemplo, de dia que possui como campo todas as demais variáveis, umidade, temperatura, nuvem. Assim, cada variável do tipo dia poderia ser ilustrada como se fosse uma ficha, como mostrado abaixo.

dia
umidade: **double**
temperatura: **double**
nuvem: **string**

Figura 21: criação de uma nova variável, que possui diversas variáveis como componentes. Tal como uma ficha.

12.1 SINTAXE DE UMA ESTRUTURA

É justamente essa organização de dados que chamamos de estrutura. Assim, uma estrutura define um novo tipo de variável, que é composta por outras variáveis, os chamados campos da estrutura.

Em algoritmo, podemos declarar uma estrutura junto com as demais variáveis, seguindo a seguinte sintaxe:

```
type dia: struct
        umidade: real
        temperatura: real
        nuvem: string
    end struct
```

Se, logo em seguida, quisermos criar uma variável do tipo dia, basta declararmos as variáveis como faríamos normalmente, tal qual se dia fosse um novo tipo de variável:

```
declare:
    natal: dia
    pascoa: dia
```

Assim, as variáveis natal e pascoa possuem campos, que são umidade, temperatura e nuvem. Para operarmos os campos de cada variável, utilizaremos o operador ponto. Veja, a seguir, alguns exemplos:

```
    natal.temperatura ← 30
    natal.nuvem← "cirrus"
    natal.umidade ←17
    pascoa.temperatura ←32
    write natal.temperatura + pascoa.temperatura
```

Portanto, basta inserir o nome da variável (que é do tipo registro), o operador ponto e o campo que se deseja operar. Todo esse conjunto se comporta exatamente como qualquer outra variável do mesmo tipo do campo, e pode ser operador da mesma forma.

12.2 ARRANJOS DE ESTRUTURAS

Porém, o que vimos até aqui não ajuda em nada na solução do problema proposto, uma vez que essa solução seria ineficiente: seria necessário criar 50 variáveis do tipo dia, e atribuir seus valores uma a uma.

Mais uma vez, podemos expandir o conceito estudado no capítulo anterior para este capítulo. Por que não criamos um arranjo de estruturas?

Depois de criar uma estrutura (aqui, consideraremos criada a estrutura dia do exemplo anterior), podemos declarar um arranjo de estruturas da seguinte forma:

```
dias[50]: dia
```

Ou seja, estamos criando um arranjo unidimensional, com 50 elementos, que são todos do tipo dia, os quais possuem, cada um, seus campos. Podemos, se desejarmos, criar arranjos de estruturas n-dimensionais, apesar de raramente haverem motivos para isso.

Por fim, segue a implementação completa de solução algorítmica que soluciona o problema proposto.

Algorithm: observações meteorológicas durante 50 dias do ano

Input: umidade do ar, temperatura e tipo de nuvem de 50 dias do ano.

Output: temperatura média nesse período

```
▸ Declaração da estrutura
        type dia: struct
                        umidade: real
                        temperatura: real
                        nuvem: string
                end struct
▸ Declaração das demais variáveis
        declare:
                dias[50]: dia
                media: real
▸ Definição do valor padrão
        media ← 0
▸ Recebimento dos valores de entrada
        for i ← 0 to 49 step 1 do
                read dias[i].umidade
                read dias[i].temperatura
                read dias[i].nuvem
        end for
▸ Cálculo da média
        for i ← 0 to 49 step 1 do
```

```
            media ← media + dias[i].temperatura
        end for
        media ← media /50
        write media
end Algorithm
```

13 PONTEIROS

Aqui, é o momento em que abordamos o conceito de ponteiro, que será utilizado no próximo capítulo, quando trataremos de sub-rotinas. Para tanto, será necessária alguma abstração, uma vez que dependemos da compreensão de como um computador real funciona.

Sabemos que as variáveis utilizadas pelo algoritmo são armazenadas em parte da memória, e que essas variáveis possuem um identificador – que permitem que elas sejam chamadas durante o algoritmo. Porém, além do identificador, cada parte da memória também possui um endereço.

Uma variável do tipo ponteiro é uma variável que guarda um endereço da na memória. Aqui, não nos interessemos por qual é a sequência de caracteres que identifica o endereço (seja ela um número real, hexareal, ou outro tipo de formatação), mas sim em qual o endereço propriamente dito.

Observe no exemplo abaixo, em que o endereço da memória é fictício – obtive-o compilando um programa simples em C++.

integer	Tipo do dado. Indica a forma como o dado será armazenado na memória.
ano_nascimento	Identificador da variável, utilizado para "chamá-la" durante o algoritmo.
1999	Valor assumido pela variável, e utilizado nas operações.
endereço: #0x61ff1c	Endereço da variável. Indica o local na memória em que a variável em questão está alocada.

Figura 22: exemplo de uma variável e suas características: tipo do dado, identificador, valor e endereço na memória.

Assim, uma variável do tipo ponteiro armazena o endereço de outra variável, mas não como um mero texto ou número, e sim como a instrução que aponta para a variável associada com aquele endereço.

Podemos declarar uma variável do tipo ponteiro com o operador ↑, seguido do tipo da variável em questão. Por exemplo:

```
declare:
    ano_nascimento: integer
    endereço_ano_nascimento: ↑integer
```

Por sua vez, podemos atribuir um endereço a uma variável do tipo ponteiro com o operador &, da seguinte forma:

```
    endereço_ano_nascimento ← &ano_nascimento
```

Entretanto, valem algumas ressalvas. Variáveis dos tipos integer, real e boolean ocupam apenas uma unidade na memória e, por isso, o operador & retorna o endereço dessa unidade.

Já arranjos dessas variáveis ocupam várias unidades da memória e, portanto, o operador & retorna apenas o endereço do primeiro elemento desses arranjos. Assim, para se obter o endereço de outros elementos do arranjo temos duas opções:

▸ Neste exemplo, temos um arranjo dado por arranjo[4]:integer e queremos o endereço da variável arranjo[2]

▸ A primeira possibilidade é:

```
endereço_arranjo ← &arranjo[0] + 2
```

▸ A segunda possibilidade é:

```
endereço_arranjo ← &arranjo[2]
```

Já para variáveis do tipo string, para estruturas ou para arranjos de mais de duas dimensões, a forma como os ponteiros funcionam foge do escopo dessa apostila, uma vez que estamos lidando com estruturas de dados mais complexas, que muitas vezes funcionam de formas distintas entre duas linguagens de programação.

Porém, em todos os casos, utilizar o operador & antes da menor unidade de variável (seja ela um campo de uma estrutura, ou um elemento de um arranjo n-dimensional) sempre irá retornar o endereço dessa unidade.

Tudo isso pode parecer muito complicado, e realmente é: até agora, estamos lidando com algoritmos escritos no papel e executados mentalmente, e é claro que nossa mente não utiliza – ao menos conscientemente – endereços de memória para as variáveis.

Buscando facilitar esse entendimento, tomei a liberdade de utilizar um programa em C++ (uma linguagem de mais baixo nível, na qual o usuário tem controle sobre a manipulação dos ponteiros) para exemplificar essa situação.

Considere o seguinte código C++, sem se preocupar com sua sintaxe. Ele cria uma variável do tipo inteiro denominada variavel, e cria um arranjo de inteiros denominado arranjo. Em seguida, atribui valores para a variável e para os elementos do arranjo, e depois faz algumas experiências com seus endereços de memória.

Note, ainda que a sintaxe para se obter um endereço é a mesma em C/C++ e em algoritmo.

```
#include <iostream>


using namespace std;
```

```
int main()
{
   // Primeiro, criamos uma variável e atribuímos um valor a ela

   int variavel;

   variavel = 49;

   cout << "O valor da variavel e " << variavel << " e seu endereco e " << &variavel << endl;
   cout << "" << endl;

   // Depois, criamos um arranjo unidimensional e atribuímos valores aos seus elementos

   int arranjo[4];

   arranjo[0] = 20;
   arranjo[1] = 30;
   arranjo[2] = 40;
   arranjo[3] = 50;
   arranjo[4] = 60;

   cout << "A operacao &arranjo retorna o endereco de seu primeiro elemento: " <<
&arranjo << endl;
   cout << "Logo, a operacao &arranjo[0] retorna exatamente a mesma coisa: " <<
&arranjo[0] << endl;
   cout << "" << endl;

   cout << "A operacao &arranjo[1] retorna o endereco do elemento seguinte: " <<
&arranjo[1] << endl;
   cout << "Logo, a operacao &arranjo[0] + 1 tambem retorna a mesma coisa que a
anterior: " << &arranjo[0] + 1 << endl;
   cout << "" << endl;
```

```
    cout << "Da mesma forma, a operacao &arranjo[3] retorna: " << &arranjo[3] << endl;

    cout << "Logo, a operacao &arranjo[0] + 3 também retorna a mesma coisa que a anterior: " << &arranjo[0] + 3 << endl;

    cout << "O qual tambem e o mesmo resultado da operacao &arranjo[1] + 2: " << &arranjo[1] + 2 << endl;

    cout << "" << endl;


    cout << "Por fim, podemos imprimir o identificador, o valor de cada elemento do arranjo, ao lado de seus enderecos de memoria: " << endl;

    for (int i = 0; i <= 4; i++){

        cout << i << "     " << arranjo[i] << "    " << &arranjo[i] << endl;

    }

}
```

Ao ser executado, ele produz a seguinte saída:

```
O valor da variavel e 49 e seu endereco e 0x61ff08


A operacao &arranjo retorna o endereco de seu primeiro elemento: 0x61fef8

Logo, a operacao &arranjo[0] retorna exatamente a mesma coisa: 0x61fef8


A operacao &arranjo[1] retorna o endereco do elemento seguinte: 0x61fefc

Logo, a operacao &arranjo[0] + 1 tambem retorna a mesma coisa que a anterior: 0x61fefc


Da mesma forma, a operacao &arranjo[3] retorna: 0x61ff04

Logo, a operacao &arranjo[0] + 3 tambÚm retorna a mesma coisa que a anterior: 0x61ff04

O qual tambem e o mesmo resultado da operacao &arranjo[1] + 2: 0x61ff04


Por fim, podemos imprimir o identificador, o valor de cada elemento do arranjo, ao lado de seus enderecos de memoria:

0     20    0x61fef8

1     30    0x61fefc

2     40    0x61ff00

3     50    0x61ff04
```

```
4    60    0x61ff08

Process returned 0 (0x0)   execution time : 0.372 s

Press any key to continue.
```

E para que exatamente ponteiros servem, além de exibir os endereços das variáveis? Na próxima unidade, estudaremos as funções e sub-rotinas, que utilizam diretamente tais conceitos.

Por enquanto, ainda há outro uso – o qual, na prática, é completamente dispensável. Lembre-se que um ponteiro não é simplesmente uma cadeia de caracteres, mas sim uma instrução que aponta para uma variável. Assim, podemos utilizar ponteiros para operar as variáveis para os quais eles apontam!

Observe um exemplo, no código abaixo.

Algorithm: exemplo do uso de ponteiros

Input: não há

Output: valor da variável, após ser alterada pelo ponteiro

▸ Declaração das variáveis

declare:

valor: **integer**

PTR_valor: ↑**integer**

▸ Atribuição do endereço da variável valor para o ponteiro

PTR_valor ← &valor ▸ O ponteiro PTR_valor aponta para a variável valor

▸ Modificação da variável valor via ponteiro

↑PTR_valor ← 241

▸ Escrita dos dados de saída

write valor

▸ O output é 241.

Assim, ao operar um ponteiro utilizando ↑ estamos, na verdade, operando a variável para o qual ele aponta.

14 SUB-ROTINAS

Até agora, o fluxo de execução dos nossos programas era quase unidirecional. No máximo, tínhamos alguns laços de repetição, mas, ainda assim, estávamos condicionados a seguir uma cadeia de comandos.

Porém, tal estilo de programar tem suas limitações. Se quiséssemos, por exemplo, executar a mesma tarefa diversas vezes, estaríamos condicionados a escrever, repetidas vezes, o mesmo trecho do código. E, caso fizéssemos, teríamos um código demasiadamente mal organizado e de difícil manutenção.

Sendo assim, as sub-rotinas caracterizam-se como trechos do código, separados da execução principal, que podem ser chamados quando necessário. Vamos, então, para um exemplo de problema que pode ser resolvido com o que sabemos até agora, mas que é de difícil execução.

"Crie um programa que trabalha com um novo tipo de dado, denominado vetor*, que é um vetor em* $\mathbb{R}^2$*, com coordenadas* x *e* y*. Em seguida, crie dois vetores, de coordenadas* $A = (1{,}3)$ *e* $B = (0{,}2)$*. Some A + B e, com isso, crie um novo vetor C. Em seguida, some C + A, e coloque esse resultado em outro ponto, D. Por fim, calcule a norma dos vetores D e A. Em seguida, exiba na tela o valor dos vetores A,B,C e D, além das normas calculadas. Além disso, considere que:*

$$(x, y) + (w, z) = (x + y, w + z)$$

$$||(x, y)|| = \sqrt{x^2 + y^2}"$$

Podemos, com algum esforço, implementar a solução para esse problema de forma linear. Observe como ela seria:

```
Algorithm: manipulações com vetores em ℝ²
Input: vetores A e B
Output: Valor dos vetores A,B,C,D, norma do vetor D e do vetor A
► Criação do tipo de dado vetor
type vetor: struct
            x: integer
            y: integer
        end struct
► Declaração das variáveis utilizadas
declare:
    A: vetor
    B: vetor
```

```
        C: vetor
        D: vetor
        normaA: real
        normaD: real
► Recebimento dos valores de cada vetor
        A.x ← 1
        A.y ← 3
        B.x ← 0
        B.y ← 2
► Cálculo da soma dos vetores
        C.x ← A.x + B.x
        C.y ← A.y + B.y
        D.x ← A.x + C.x
        D.y ← A.y + Cy
► Cálculo da norma dos vetores
        normaA ← raiz((A.x)^2 +( A.y)^2)
        normaD ← raiz((D.x)^2 +( D.y)^2)
► Escrita dos dados de saída
        write "(", A.x, ",", "A.y, ")"
        write "(", B.x, ",", "B.y, ")"
        write "(", C.x, ",", "C.y, ")"
        write "(", D.x, ",", "D.y, ")"
        write normaA
        wrtie normaD

end Algorithm
```

Se, lendo o algoritmo, você não teve a sensação de exaustão, tente escrevê-lo. Existem diversas tarefas que são repetidas e que poderiam ser especificadas apenas uma vez, tais como o recebimento das coordenadas de cada vetor, a soma, o cálculo da norma e a escrita dos dados. Que tal definirmos uma sub-rotina para cada uma dessas tarefas?

14.1 PROCEDIMENTOS

A sub-rotina de escrita dos dados apenas recebe um vetor e escreve, para o usuário, o valor dos campos x e y, além dos parênteses presentes na exibição. Assim, ela apenas executa uma sequência de passos, mas não retorna nenhum tipo de valor.

O mesmo vale para a sub-rotina de recebimento das coordenadas de cada vetor: não há nenhum tipo de valor retornado, a sub-rotina apenas recebe os valores de cada coordenada, o vetor que terá seus campos alterados, e faz tal alteração.

A essas sub-rotinas, que não tem nenhum tipo de valor de retorno, chamamos de procedimento. Além disso, todo procedimento tem um indicador (que é basicamente o nome do procedimento, ou seja, como ele é chamado) e os parâmetros que ele recebe (ou seja, as variáveis que uma sub-rotina recebe para realizar a sequência de passos).

Por exemplo, o procedimento que exibe o vetor na tela pode ser chamado de exibir, e receber, como parâmetro, uma variável do tipo vetor chamada exibir_v. Observe, abaixo, como ele seria implementado utilizando a sintaxe específica.

```
procedure exibir (exibir_v: vetor)
        ►Exibe o valor de um vetor na tela
        write "(", exibir_v.x, ",", exibir_v.y, ")"
end procedure
```

Tendo criado esse procedimento, podemos chama-lo em qualquer momento do código da mesma forma que chamaríamos as funções estudadas no capítulo 5. Assim, nossa parte final do algoritmo desenvolvido poderia ser substituída por:

```
► Escrita dos dados de saída
        exibir(A)
        exibir(B)
        exibir(C)
        exibir(D)
```

Note que, nesta sub-rotina, o vetor que é passado como parâmetro não necessita ser alterado. Assim, a passagem realizada envolve apenas a cópia do valor que o vetor contém, em um processo é chamado passagem por cópia.

Assim, sempre que os parâmetros forem passados pelo identificador da variável, estamos apenas copiando os valores da variável. Logo, qualquer alteração

realizada nessa variável dentro da sub-rotina irá apenas alterar o valor copiado, e não a variável em si.

A próxima sub-rotina também é um procedimento, e se chamará receber_v. Como parâmetros, ela deverá receber os dois valores atribuídos a cada uma das coordenadas, que podem ser passados por cópia (uma vez que não serão alterados) e o vetor que receberá as duas coordenadas. Este, por sua vez, será alterado durante a sub-rotina e, portanto, não pode ser passado por cópia.

É aqui que introduzimos um conceito novo, a passagem por referência. Ela é feita passando-se o endereço de memória na qual a variável que queremos alterar está alocada. Certamente, esse processo lhe recorda o conceito de ponteiros, que estudamos no capítulo anterior.

Assim, é justamente isso que faremos. Uma sub-rotina receberá o endereço para uma variável dentro de uma variável do tipo ponteiro, a qual será operada dentro da sub-rotina.

No entanto, não usaremos a mesma sintaxe vista no capítulo anterior, no qual ponteiros são indicados por ↑ e endereços são passados com &. Optamos por tal convenção porque ponteiros são particularidades de cada linguagem de programação, bem como a forma como as variáveis dos tipos string, os arranjos n-dimensionais e as estruturas são armazenadas na memória.

Logo, utilizá-los em algoritmo poderia causar confusão ao estudante, e, ao o fazermos, estaríamos condicionando nosso algoritmo a estruturação de dados utilizada em uma linguagem específica, que não é o que procuramos: algoritmos precisam ser genéricos.

Assim, sempre que uma sub-rotina receber um parâmetro por referência (independentemente de seu tipo), a variável que o recebe será precedida pela palavra var. Veja abaixo.

```
procedure receber (val_x: integer, val_y: integer, var receber_v: vetor)
        ►Recebe as coordenadas x e y e as coloca em um vetor.
        receber_v.x ← val_x
        receber_v.y ←val_y
end procedure
```

Já para chamar a sub-rotina, continuamos a indicar todos os parâmetros da mesma forma: apenas pelo seu identificador. Isso vale para tipos oriundos de estruturas, arranjos n-dimensionais, strings ou qualquer outro tipo de dado.

```
► Recebimento dos valores de cada vetor
        receber(1, 3, A)
        receber(0, 2, B)
```

Podemos, também, criar a sub-rotina para o cálculo da norma de um vetor utilizando passagem por referência. Assim, o procedimento receberia como parâmetros duas variáveis: o vetor, por cópia, e a variável que receberá a norma do vetor, por referência. O resultado seria:

```
procedure norma_ref (calc_v: vetor, var norma: integer)
        ►Calcula a norma de um vetor, e a coloca em uma variável
        norma ← raiz((calc_v.x)^2 + (calc_v.y)^2)
end procedure
```

No código principal, teríamos:

```
► Cálculo da norma dos vetores
      norma_ref(A, normaA)
      norma_ref(B, normaB)
```

14.2 FUNÇÕES

A passagem de valores por referência em um procedimento é útil quando queremos alterar mais de uma variável dentro de nossa sub-rotina. Entretanto, quando a sub-rotina altera apenas uma variável, pode ser mais fácil criar uma função, ao invés de um procedimento.

Ao contrário dos procedimentos, funções são um conjunto de passos que, ao final do fluxo de execução, retorna uma variável (e apenas uma) para o código principal.

Assim, uma função é descrita pelo seu identificador (da mesma forma que um procedimento), pelos parâmetros (que podem ser passados por cópia ou por referência, também da mesma forma que em um procedimento) e pelo seu tipo de retorno, que pode ser um dos tipos padrões, um arranjo n-dimensional ou uma estrutura.

Por exemplo, vamos implementar a sub-rotina que calcula a norma dos vetores como uma função. Isso será feito por meio da seguinte sintaxe:

```
function norma_func(calc_v: vetor): real
        ►Calcula a norma de um vetor, e a retorna
        declare:
                norma: real
        norma ← raiz((calc_v.x)^2 + (calc_v.y)^2)
```

```
        return norma
end function
```

Toda função tem um tipo de retorno, que corresponde ao tipo da variável (neste caso, da variável norma) que será retornada na função. É possível que, dentro da função, haja uma condicional, a qual retorna variáveis diferentes conforme o fluxo de execução. Porém, todas elas devem ser, obrigatoriamente, do mesmo tipo.

Além disso, é necessário introduzir o conceito de escopo, que vale tanto para procedimentos quanto para funções: variáveis declaradas dentro de uma sub-rotina podem ser utilizadas apenas dentro dessa sub-rotina. Se uma variável de mesmo nome for declarada fora do escopo da sub-rotina, ela possuirá outro endereço e, efetivamente, será outra variável.

Já no código principal, podemos igualar qualquer variável do mesmo tipo de retorno da função com a própria execução da função, da seguinte forma:

```
► Cálculo da norma dos vetores
        normaA ← norma_func(A)
        normaD ← norma_func(D)
```

Assim, podemos implementar, de forma análoga, a função para a soma de dois vetores. Ela será:

```
function soma(V1: vetor, V2:vetor): vetor
        ►Calcula a soma de dois vetores, e a retorna
        declare:
                r_soma: vetor
        r_soma.x ← V1.x + V2.x
        r_soma.y ← V1.y + V2.y

        return r_soma
end function
```

Da mesma forma, a função pode ser implementada no código principal da seguinte maneira:

```
► Cálculo da soma dos vetores
        C ← soma(A,B)
        D ← soma(A,C)
```

Assim, podemos encerrar essa seção apresentando o algoritmo reescrito, utilizando sub-rotinas.

```
Algorithm: manipulações com vetores em ℝ²
Input: vetores A e B
Output: Valor dos vetores A,B,C,D, norma do vetor D e do vetor A
► Criação do tipo de dado vetor
type vetor: struct
        x: integer
        y: integer
    end struct
► Declaração das sub-rotinas
procedure exibir (exibir_v: vetor)
    ►Exibe o valor de um vetor na tela
    write "(", exibir_v.x, ",", exibir_v.y, ")"
end procedure
procedure receber (val_x: integer, val_y: integer, var vetor_rec: vetor)
    ►Recebe as coordenadas x e y e as coloca em um vetor.
    vetor_rec.x ← val_x
    vetor_rec.y ←val_y
end procedure
function norma_func(vetor_calc: vetor): real
    ►Calcula a norma de um vetor, e a retorna
    declare:
        norma: real
    norma ← raiz((vetor_calc.x)^2 + (vetor_calc.y)^2)
    return norma
end function
function soma(V1: vetor, V2:vetor): vetor
    ►Calcula a soma de dois vetores, e a retorna
    declare:
        r_soma: vetor
    r_soma.x ← V1.x + V2.x
```

```
        r_soma.y ← V1.y + V2.y


        return r_soma
end function
► Declaração das variáveis utilizadas
declare:
        A: vetor
        B: vetor
        C: vetor
        D: vetor
        normaA: real
        normaD: real
► Recebimento dos valores de cada vetor
        receber(1, 3, A)
        receber(0, 2, B)
► Cálculo da soma dos vetores
        C ← soma(A,B)
        D ← soma(A,C)
► Cálculo da norma dos vetores
        normaA ← norma_func(A)
        normaD ← norma_func(D)
► Escrita dos dados de saída
        exibir(A)
        exibir(B)
        exibir(C)
        exibir(D)
        write normaA
        wrtie normaD
end Algorithm
```

Em linhas gerais, siga as seguintes diretrizes para realizar a passagem de valores para uma sub-rotina.

1. Na construção da sub-rotina, variáveis que recebem valores por referência devem ser declaradas com a palavra var antes do nome da variável. Variáveis que recebem valores por cópia devem ser declaradas apenas pelo seu identificador e tipo.

2. Dentro da sub-rotina, todas as variáveis são operadas apenas pelo seu identificador, campos e índices, da mesma forma que no código principal.

3. Ao chamar uma sub-rotina, as variáveis de qualquer tipo devem ser passadas por seu identificador, seja por cópia ou por referência.

4. Procedimentos e funções podem receber qualquer número de valores, seja por referência ou por cópia.

5. Procedimentos não possuem nenhum valor de retorno.

6. Funções possuem apenas um valor de retorno.

7. Uma função pode retornar qualquer tipo de dado.

8. Variáveis criadas dentro de uma sub-rotina só podem ser utilizadas dentro dessa sub-rotina.

9. Compete ao programador, ao implementar o algoritmo em uma dada linguagem, utilizar os ponteiros e endereços de forma adequada para realizar a passagem por referência.

15 CONSTANTES GLOBAIS

Esse capítulo é, provavelmente, um dos mais simples deste curso. Para tanto, considere o problema discutido no capítulo 11:

"Construa um programa que receba as três notas de um aluno em uma disciplina. Em seguida, o programa deve imprimir um boletim, contendo cada uma das notas e a média final."

Ao utilizar vetores, concluímos que uma solução adequada para o problema em questão seria:

```
Algorithm: boletim
Input: notas de um aluno em 3 provas.
Output: boletim contendo nota de cada prova e sua média final
► Declaração das variáveis utilizadas
declare:
        notas[3]: real
        media: real

► Leitura das notas
for i → 0 to 2 step 1 do
        read notas[i]
        media ← notas[i]
end for
► Determinação da média
media ← media / 3
► Escrita dos resultados
for i → 0 to 2 step 1 do
        write notas[i]
end for
write media
end Algorithm
```

Entretanto, suponha que, agora, temos 6 notas, ao invés de 3. Poderíamos adaptar esse algoritmo alterando, pontualmente, cada passo que é impactado pela

quantidade de notas. Ao todo, faríamos quatro alterações, e esse número seria bem maior em algoritmos mais complexos.

Assim, visando facilitar a manutenção e eventual alteração do código, podemos definir uma "variável" chamada total_notas, do tipo inteiro, que recebe a quantidade de notas. Essa "variável" poderia, nas demais partes do código, ser referenciada, ou ter seu valor copiado e operado conforme necessário, mas jamais poderia ser alterada.

A esse tipo de "variável" chamamos constante global, e ela pode ser declarada da seguinte forma:

```
define:
        total_notas: integer  6
```

Em que total_notas corresponde ao identificador da constante global, integer corresponde ao seu tipo, e 6 corresponde ao seu valor. Assim, no decorrer do código, toda ocorrência com o nome total_notas deve ser substituída pelo valor que a constante global possui, no caso, 6.

Assim, a constante global poderia ser inserida no código da seguinte forma:

```
Algorithm: boletim
Input: notas de um aluno em 3 provas.
Output: boletim contendo nota de cada prova e sua média final
► Declaração das variáveis utilizadas
define:
        total_notas: integer  6
declare:
        notas[total_notas]: real
        media: real

► Leitura das notas
for i → 0 to (total_notas – 1) step 1 do
        read notas[i]
        media ← notas[i]
end for
► Determinação da média
media ← media / total_notas
► Escrita dos resultados
```

```
for i → 0 to (total_notas – 1) step 1 do
        write notas[i]
end for
write media
end Algorithm
```

Assim, a alteração do código tornou-se muito mais fácil: basta alterarmos um único valor, que a mudança se propagará no momento da execução do código. Vale lembrar que tal alteração deve ser feita manualmente pelo programador, antes da execução.

Além disso, constantes globais, conforme o nome indica, possuem escopo global: podem ser utilizadas no código principal ou em sub-rotinas.

16 CONSIDERAÇÕES FINAIS

Assim, encerramos nosso curso de lógica de programação. Com os conhecimentos aqui aprendidos, você certamente terá muito mais facilidade ao se iniciar em alguma linguagem, até mesmo as de baixo nível, nas quais os tópicos dos capítulos finais se fazem presentes.

Se você é um entusiasta na computação, ou um estudante de outra área, recomendo avançar seus estudos para alguma linguagem de programação de alto nível, tal como Python.

Já se você é um estudante de computação, é provável que não tenha a mesma sorte, e deva prosseguir seus estudos, por força do curso, para uma linguagem de mais baixo nível, tal como a linguagem C. Ainda assim, não se preocupe: boa parte do que estudamos nos últimos capítulos será fundamental para implementar estruturas como ponteiros e passagens por referência nessas linguagens.

Espero que este curso tenha sido proveitoso, até a próxima!

UM BREVE COMENTÁRIO...

Espero que essa apostila lhe tenha sido útil. Espero, também, que ela tenha chegado as suas mãos da forma que foi feita para chegar: gratuitamente.

Este mateiral faz parte de um projeto que eu, estudante de Engenharia da Computação, mantenho em meu canal no YouTube, visando apresentar a Computação de forma mais simples e voltada para o público leigo.

Se você considera meu trabalho merecedor de qualquer tipo de recompensa, aceito contribuições em criptomoedas, PayPal e Patreon. Veja detalhes em www.fabricadenoobs.com.br/colaborar, ou por meio do QR Code abaixo.

Muito obrigado!